ABSTRACT

EL-HAJ MAHMOUD, ALI AHMAD. Hard-Real-Time Multithreading: A Combined Micro-architectural and Scheduling Approach. (Under the direction of Dr. Eric Rotenberg).

Simultaneous Multithreading (SMT) enables fine-grain resource sharing of a single superscalar processor among multiple tasks, improving cost-performance. However, SMT cannot be safely exploited in hard-real-time systems. These systems require analytical frameworks for making worst-case performance guarantees. SMT violates simplifying assumptions for deriving worst-case execution times (WCET) of tasks. Classic real-time theory uses *single-task WCET analysis*, where a task is assumed to have access to dedicated processor resources, hence, its WCET can be derived independent of its task-set context. This is not true for SMT, where tasks interfere due to resource sharing. Modeling interference requires whole task-set WCET analysis, but this approach is futile since co-scheduled tasks vary and compete for resources arbitrarily. Thus, formally proving real-time guarantees for SMT is intractable.

This dissertation proposes *flexible interference-free multithreading*. Interference-free partitioning guarantees that the performance of a single task is not affected by its workload context (hence, preserving single-task WCET analysis), while flexible resource sharing emulates fine-grain resource sharing of SMT to achieve similar cost-performance efficiency.

The *Real-time Virtual Multiprocessor* (RVMP) paradigm virtualizes a single superscalar processor into multiple interference-free different-sized virtual processors. This provides a flexible spatial dimension. In the time dimension, the number and sizes of virtual processors can be rapidly reconfigured. A simple real-time scheduling approach concentrates scheduling within a small time interval (the “round”), producing a simple repeating space/time schedule that orchestrates virtualization.

Worst-case schedulability experiments show that more task-sets are provably schedulable on RVMP than on conventional rigid multiprocessors with equal aggregate resources, and the advantage only intensifies with more demanding task-sets. Run-time experiments show RVMP’s statically-controlled coarser-grain resource sharing is as effective as unsafe SMT, and provides a real-time formalism that SMT does not currently provide.

RVMP’s round-based scheduling enables other optimizations for safely improving performance even more. A framework is developed on top of RVMP to safely, tractably, and tightly bound overlap between computation and memory accesses of different tasks to improve worst-case performance. This framework captures the throughput gain of dynamic switch-on-event multithreading, but in a way that is compatible with hard-real-time formalism.

HARD-REAL-TIME MULTITHREADING:
A COMBINED MICROARCHITECTURAL AND SCHEDULING APPROACH

by

ALI AHMAD EL-HAJ MAHMOUD

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

COMPUTER ENGINEERING

Raleigh

2006

APPROVED BY:

Dr. Eric Rotenberg Chair of Advisory Committee	Dr. Thomas M. Conte
Dr. Alexander G. Dean	Dr. Frank Mueller

Dedication

To my loving parents . . .

your constant encouragement . . .

gave me strength to carry on . . .

إلىٰ والِدَيّ العَزيزَينْ ...

دُونَ كلِّ ما بذَلتُما ...

لَمْ أَكُنْ لِأُواصِلَ حَتّىٰ النَّهايةُ ...

Biography

Ali El-Haj-Mahmoud was born in the Lebanese capital Beirut, to Dr. Ahmad El-Haj-Mahmoud and Mrs. Souad Alamani. He moved at the age of two weeks with his parents to the small town of Ras Al-Khaimah, United Arab Emirates, where he grew up and received his schooling. In 1996, Ali passed the United Arab Emirates General Secondary School Examination (Scientific Section) with High Distinction, ranking 7th on the whole country. Then, he moved back to Beirut, Lebanon to attend the American University of Beirut, majoring in Computer and Communication Engineering. He received his Bachelor's of Engineering with Distinction in 2001. After graduation, Ali moved to the United States to pursue his graduate studies in Computer Engineering at North Carolina State University. He received the Master's degree in 2002 and the Doctor of Philosophy degree (under the direction of Dr. Eric Rotenberg) in 2006. During his graduate career, Ali was invited to join Phi Kappa Phi and Eta Kappa Nu, the two most selective and prestigious honors societies, in recognition of his academic achievements.

Upon completing his Ph.D., Ali is moving back to Dubai, United Arab Emirates, to join the American University in Dubai as an Assistant Professor of Computer Engineering.

Acknowledgments

Pursuing a Ph.D. degree can be a very overwhelming and life-changing experience. During the five years of my graduate career, I've been through many periods of stress and uncertainty. If it wasn't for the help and support of many wonderful people, it wouldn't have been possible for me to be at this stage today.

First and foremost, I'm very grateful and indebted to my parents Dr. Ahmad El-Haj-Mahmoud and Mrs. Souad Alamani, my elder brother Samer, and my sweet little sisters Rana and Hazar. Their unconditional love, devotion, and support (emotional, financial, and everything else in between) gave me hope when I needed it most. Without them backing me up throughout my graduate career (and my life), I wouldn't be writing this today. No matter what I do or say, I won't be able to pay them back what they really deserve.

I've been very lucky and privileged to have Dr. Eric Rotenberg not only as my Ph.D. advisor, but also as a mentor and a friend. Eric has taught me by example new meanings of commitment, hard work, and enthusiasm, all while striving for excellence that falls just short of the impossible perfection. The most valuable lesson I learned from Eric about research is the importance of looking beyond details to understand the "crux" of the problem, and then, the importance of writing as a craft to clearly convey the crux to the reader. Eric gave me the opportunity to work on a challenging and rewarding problem, and at the same time allowed me enough freedom to pursue my personal research interests. If I had to do it all over again, I wouldn't want a different advisor.

I'm very thankful to my committee members: Dr. Thomas Conte, Dr. Alexander Dean, and Dr. Frank Mueller. Their feedback and comments were very helpful in shaping this dissertation. I'm specially grateful to Dr. Conte and Dr. Dean for all their help and advice throughout my graduate career at NC State.

My decision to pursue a graduate career was very much influenced by Dr. Ayman Kayssi at the American University of Beirut. Dr. Kayssi introduced me to the world of academic research during my undergraduate study, which helped me confirm my desire to choose this path. Even after I joined NC State, Dr. Kayssi stayed in touch, giving his encouragement and support during much needed times.

During my Ph.D. years at the CESR lab, I've met a great group of people, who helped make my stay here more fruitful and enjoyable. I want to thank Mazen Kharbutli, Aravindh Anantaraman, Muawya Al-Otoom, Ahmed Al-Zawawi, Sailashri Parthasarathy, Vimal Reddy, Ravi Venkatesan, Hashem Hashemi, Balaji Iyer, Saurabh Sharma, Paul Bryan, Chad Rosier, Zach Purser, Nikhil Gupta, and Prakash Ramrakhyani for all their help and company.

Being my housemate and office-mate, Mazen Kharbutli (a.k.a. the "`malloc()` man" or "DJ Mazen") was more of a brother than a friend. There are many things to thank Mazen for, in fact, too many to mention here. However, I'd like to mention just a few memorable moments over the years, including ritual dinners at Denny's after paper submission deadlines, AC Milan vs. Barcelona soccer game in Washington DC, Orlando theme parks and acrophobia, the Vitara tragic departure, “don’t make me angry, you wouldn’t like me when I’m angry”, “trust in me, have no fear” (and go to the wrong airport while doing that!), and of course, “it’s just chaos”.

I’m especially thankful to Aravindh Anantaraman, my “co-conspirator and partner in crime”, for always listening to my questions and ideas, providing excellent feedback, and always being there the nights of paper submission deadlines to help in the final crunch that makes ends meet. Working on closely-related research problems, I had the opportunity to interact with and learn a lot from Aravindh, which greatly helped me grow technically and personally. Thank you Aravindh for being such a great colleague and friend, and good luck at Intel (waving hand!).

I have to acknowledge Ahmed Al-Zawawi (the if-there-is-a-deal-on-anything-under-the-sun-then-he-got-it guy) for being the “idea hamster” he is, and for readily providing a lot of good starting points that ended up in this dissertation. A special thank you is due to Vimal Reddy for being our instant C++ reference manual. I want to thank Sailashri Parthasarathy (the “proppa proppa chill-pill” and expert in “cooking iced tea”) for always cheering everyone up and reminding everyone to enjoy the small pleasures of life while they last. I also want to thank Saila for the yummy “homemade” truffles that I never actually received!

A special thank you is due to Sandy Bronson, our CESR Administrative Assistant, for taking care of all the paperwork and making our lives, the grad students, a bit easier.

I was very fortunate to have a wonderful group of friends who became my second family away from home. I’m very indebted for all their help settling down in Raleigh, overcoming the initial homesickness, and living the day-to-day life. I want to specifically thank the “1009 Carlton Ave” guys: Amro Bohsali (the big guy), Samer El-Housseini, Wissam Fazaa (ana zah2aan!!), Mazen Kharbutli, and Mohammad Darwish (well, technically, he wasn't our housemate, but he spent more time with us in Raleigh than at his place in Orlando!), in addition to my current housemates and friends: Mahmoud "MC" Chehab, Tareq Ghaith, Muawya Al-Otoom, and Monther Aldwairi. Some moments I will always remember, and will sure bring a smile to my face: ever-lasting card games, lake Johnson "trips", "you've got to push the button!", "educational sessions" at TGI Fridays, late night boredom and grilling at 4 AM, Honda (or Bohs) Effect, darts tournaments, the "Butcher Shop" project, road trips along the east coast, Wolfpack basketball wins and losses, "ya shabab! badna ennazzemha!!", the ever famous (and at times funny) "zmek" dog joke (Sorry Amro, I have to put this one in!), and of course, "what happens in Raleigh, stays in Raleigh" (I hope I'm not already breaking this sacred covenant!), among too many others to mention. I will fondly cherish those days.

I want to also thank my friends from the undergraduate days at AUB, the "Ba7sh" gang: Mohamad Zeidan, Mohammad Darwish, Talal Aranout, Mohammad Al-Sawda, Ziad Al-Bawab, Peter Al-Hokayem, Samer El-Housseini, Wissam Fazaa, and Bassem Abdouni. If I want to mention all what you did for me, the list will just go on and on. Thank you guys for everything.

I cannot but separately thank my "Brother in Arms", Imad Ferzli for all the support since our undergraduate days at AUB. I will always remember all the great discussions we had – instead of actually doing work – about politics, religion, philosophy, music, and movies, in addition to all the crazy technology startup ideas ("Maameltein" Silicon Valley will definitely NOT see the light!).

I'm very grateful to Ghassan Chehab and Ali Termos for being very good friends and mentors, especially when I first came to Raleigh. I always knew where to go when I needed a listening ear, a wise advice, or just a good ol' intellectual chat.

I want to thank Jorge Cham for drawing his hilarious comic strip: "PhD: Piled Higher and Deeper" about the life (or the lack thereof) in graduate school. PhD Comics was a great source of inspiration, hope, and procrastination (!) throughout my graduate school years. At times of depression, it is just very relieving to know that many other people feel the same stress, anxiety, and uncertainty as you. If Mike Slackenerny could do it, then everybody can!

Last, but definitely not the least, a special thank you is extended to Rima Khalaf. If there is one person who re-shaped my life, changed the way I think, and transformed who I am, then that person would be Rima. No words can express my gratitude to you, Rima. Thank you for believing in me even when I didn't believe in myself. Thank you for helping me see the world in colors.

This research was supported in part by NSF CAREER grant CCR-0092832, NSF grants CCR-0207785, CCR-0208581 and CCR-0310860, and generous funding and equipment donations from Intel and Ubicom. Any opinions, findings, and conclusions or recommendations expressed in this publication herein are those of the author and do not necessarily represent the views of the National Science Foundation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation develops a hard-real-time-compatible simultaneous multithreading (SMT) framework. To my knowledge, this is the first such framework to support safe *hard-real-time* scheduling on an SMT processor.

SMT was proposed in the context of high-performance general-purpose computing, to enable fine-grain resource sharing of a single superscalar processor among multiple tasks, yielding good cost-performance. It is compelling to consider SMT in the context of real-time embedded systems because these systems run task-sets with multiple periodic tasks, naturally providing thread-level parallelism (TLP) and the opportunity to apply SMT. As the number of tasks grow in complex embedded systems such as automotive systems, avionics, *etc.*, SMT becomes attractive as a means to consolidate hardware and software into a few or even one high-performance multithreaded processor.

Unfortunately, SMT is inherently incompatible with hard-real-time systems, because SMT makes it intractable to guarantee the timing correctness required by these systems.

Classic real-time theory guarantees timing correctness hierarchically, in two steps. First, worst-case execution times (WCET) of tasks are derived via **single-task WCET analysis**: WCET timing analysis is performed for each task independently of other tasks, with the assumption that a task has access to the whole processor or at least dedicated processor resources. Second, using tasks' WCETs as inputs, a real-time scheduling test is applied *a priori* to determine whether or not the task-set is feasibly schedulable as a whole on the processor. Usually, the test determines the schedulability of the task-set without actually constructing a schedule.

SMT is incompatible with this theoretical framework because SMT violates single-task WCET analysis. *Since multiple tasks execute simultaneously, sharing the processor's resources, the WCET of a task can no longer be derived independent of its workload context.* On the contrary, the WCET of each task must now be derived taking into account interference with other tasks in the task-set. Alternatively, forgoing the classic hierarchical WCET/scheduling test approach altogether, we could attempt to statically construct a valid schedule that meets all deadlines. Either strategy is intractable. Tasks have different periods, thus, co-scheduled task combinations and the degree of overlap among co-scheduled tasks will vary over time. Moreover, instructions from different tasks compete for pipeline resources in a highly unpredictable manner. All of these issues make concluding hard-real-time schedulability on SMT extremely difficult.

To achieve hard-real-time-compatible simultaneous multithreading, the key idea is to provide *flexible (adjustable) resource sharing* among tasks, yet guarantee *interference-free* execution among tasks, where the performance of a single task is not affected by its workload context. With flexible interference-free partitioning, we preserve single-task WCET analysis and hierarchical schedulability analysis, and at the same time emulate fine-grain resource sharing of SMT to achieve similar cost-performance efficiency.

The *Real-time Virtual Multiprocessor* (RVMP) paradigm [32, 31] aims at meeting the contradicting goals of *high-performance* demanded by high-end embedded applications and *analyzability* needed to guarantee the timing correctness of safety-critical real-time systems.

My dissertation puts forth two theses, the first is specific to RVMP and the second is a generalization of the overall methodology that RVMP is an application of.

- *Flexible interference-free partitioning provides the necessary framework to safely and tractably schedule hard-real-time tasks on a simultaneous multithreading processor*. Such a framework combines the analyzability required by real-time systems (by preserving single-task WCET analysis) with the cost-performance benefits of dynamic SMT.

- *Co-designing processor architecture and real-time scheduling leads to better real-time performance*. By designing an architecture with real-time performance guarantees as a main goal, and then exposing the features of this architecture to real-time scheduling, major improvements in hard-real-time performance can be achieved.

1.1 Motivation: Performance vs. Analyzability in Real-Time Embedded Systems

Driven by customers' insatiability for new features in highly competitive markets, high-end embedded processors are striving to deliver higher performance within stringent constraints of low cost, low power consumption, and small footprint area dictated by ever decreasing form factors. To meet these higher performance targets, high-end embedded processors have started inheriting performance-enhancing techniques from their high-performance desktop counterparts. Features like deep pipelining, branch prediction, multithreading, and even out-of-order execution are starting to find ground in the embedded processor domain. For example, ARM11-derived processors have an 8-stage pipeline with dynamic branch prediction and caches [23], Ubicom's IP3023 supports 8 hardware threads [90] and MIPS 34K supports 5 [78], and IBM's embedded PowerPC 750 [45] is a dynamically-scheduled 2-way superscalar processor.

Unfortunately, dynamic performance-enhancing techniques complicate the design process of an important class of embedded systems, *hard-real-time systems*. A hard-real-time system typically runs a set of periodic tasks (collectively called a *task-set*). Each task repeats at fixed time intervals equal to the period of the task (Figure 1-1 shows an example task-set with two periodic tasks A and B). Each task has a deadline, often equal to its period [63]. That is, an instance of a task (*e.g.*, $A1$ in Figure 1-1) must finish execution before the next instance of the task (*e.g.*, $A2$ in Figure 1-1) is released. Guaranteeing this criterion for all tasks in the task-set guarantees the task-set is schedulable as a whole. Schedulability of a task-set must be proven or disproven *a priori* (*i.e.*, statically, before running the system), usually using the

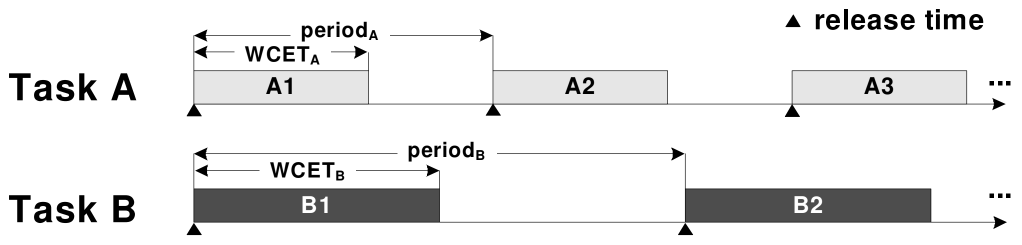
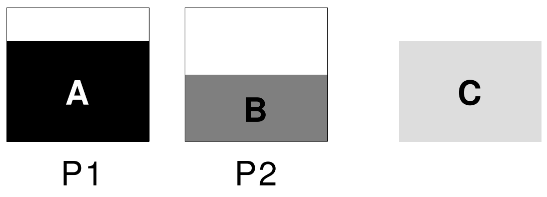


Figure 1-1. Periodic task model.

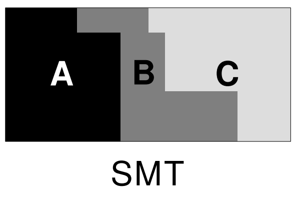
concept of processor utilization [62, 63]. Processor utilization by a task is the fraction of time the processor spends executing instances of that task, and is defined as the task's worst-case execution time (WCET) divided by its period ($U = \frac{WCET}{period}$). For example, according to the earliest-deadline-first (EDF) scheduling policy [62], for a task-set to be schedulable, the total processor utilization by all tasks must be less than or equal to 1 (*i.e.*, $\sum_i \frac{WCET_i}{period_i} \leq 1$). Despite advances in worst-case timing analysis, in practice, deriving tight and safe (provably never exceeded) WCETs of tasks on processors with dynamic branch prediction, caches, out-of-order execution, *etc.*, is intractable [*e.g.*,4]. Since availability of safe WCETs is a major requirement for formal schedulability analysis, dynamic microarchitecture features significantly complicate, and even undermine, the design process of real-time systems [39, 34, 10, 4] (or systems that must support both real-time and non-real-time applications). Often times, such features are excluded from these systems, sacrificing performance for the sake of analyzability.

Higher performance can be achieved, without sacrificing analyzability, by increasing the frequency of a simple processor via deeper pipelining. Increasing the processor's frequency results in higher power consumption [*e.g.*,41], which is undesirable for battery-operated embedded devices. In addition to power issues, deep pipelines increase design complexity. Finally, memory performance is increasing at a slower rate than processor frequency. As a result, the processor-memory speed gap that plagues high-performance processors will resurface again in embedded systems [78, 84], resulting in lower than expected gains in performance from increased frequency. Due to power, complexity, and eventual diminishing performance returns (exacerbated by branch mispredictions, memory latency, and pipeline overheads), deep pipelining is not scalable in the long run.

Using multiple simple processors (*i.e.*, for which WCET analysis can be performed) is another option for achieving high performance. Multiple simple processors is an attractive solution because of the natural availability of multiple tasks in typical real-time embedded systems, capitalizing on thread-level parallelism (TLP) as a source of performance. However, this performance increase comes at a substantial cost in terms of die area, power consumption, and footprint. Moreover, the rigid and uniform partitioning of resources among multiple processors leads to load-balance problems, which may cause demanding task-sets to be artificially unschedulable. That is, sufficient resources may be available in aggregate, but individual tasks cannot be spread across multiple processors. Alternatively, the system could be over-designed with more processors to compensate, increasing cost even more. For example, Figure 1-2(a) shows a task-set consisting of three tasks (A, B, and C) to be scheduled on two processors ($P1$ and $P2$). The "height" of a task in the figure represents its fractional utilization of the processor. Although $P1$ and $P2$ have enough resources *combined* to meet the computational demand of the task-set, the rigid partitioning of resources makes the task-set unschedulable (task C cannot be divided between the two processors). A third processor is required to schedule the task-set.



(a) Rigid multiprocessor.



(b) Flexible SMT.

Figure 1-2. Resource partitioning.

A more flexible substrate for resource sharing is preferred, to better utilize aggregate resources and improve the cost-performance metric. Simultaneous Multithreading (SMT) [99, 89] meets a similar need in high-performance general-purpose processors, enabling fine-grain resource sharing among multiple threads for higher overall system throughput. For example, in Figure 1-2(b), a single SMT processor with equal aggregate resources to $P1$ and $P2$ of Figure 1-2(a) can successfully run the same task-set, avoiding load-balance issues of multiple processors.

However, SMT undermines the analytical foundation of hard-real-time scheduling. Because there is interference among simultaneous tasks, the WCET of a task must now be derived in the context of other tasks in the task-set. This is contrary to conventional worst-case timing analysis, which assumes a task runs alone on the processor, and as such derives WCETs of tasks separately. Moreover, deriving WCETs of multiple tasks running together on an SMT processor is intractable. Since tasks have different periods, specific task combinations vary over time, as does the amount by which co-scheduled tasks overlap. Even if we know which tasks are running and which of their regions overlap, instructions from different tasks dynamically compete for shared processor resources. For the above reasons, SMT is incompatible with proving hard-real-time guarantees.

1.2 Hard-Real-Time Multithreading: Combining High-Performance and Real-Time Analyzability

This research aims at reconciling the trade-off between performance and analyzability by introducing a multithreading substrate compatible with formally proving real-time guarantees. The proposed *Real-time Virtual Multiprocessor* (RVMP) alleviates the issues that make dynamic SMT incompatible with real-time systems design (most importantly, interference among threads), while emulating the flexibility of SMT's fine-grain resource sharing. As a result, RVMP guarantees tasks' deadlines while achieving SMT-like throughput for the system as a whole.

RVMP consists of an analyzable high-performance microarchitecture and a simple real-time scheduling policy. An overview of these two components and the interaction between them is presented next.

1.2.1 RVMP Microarchitecture Overview

The novel RVMP architecture combines the analyzability of multiple dedicated processors with the flexible resource sharing (hence higher performance and favorable cost-performance) of SMT. We propose a highly reconfigurable multithreaded superscalar processor that provides two levels of flexibility, in *space* and *time*. In the space dimension, the processor's resources can be arbitrarily partitioned to create multiple dedicated *virtual processors* (VP), with possibly different performance levels according to the resource partitioning. Multiple tasks execute at the same time, one on each partition, *without interfering with each other*. Interference-free partitions achieve the necessary isolation for analyzability, like a conventional multiprocessor. Yet, because different-sized partitions can be carved out of the aggregate resources of the single superscalar processor underneath, we overcome schedulability limitations of multiple equal-sized processors. Superscalar "ways" (for example, there are 4 ways in a 4-way superscalar processor) present a natural resource partitioning strategy. For example, Figure 1-3(a) shows two interference-free partitions or VPs, one composed of 1 way and the other of 3 ways. In the time dimension, the resource partitions can be rapidly reconfigured, even every cycle, fluidly changing the number and size of partitions if so desired. For example, Figure 1-3(b) shows the same two interference-free partitions being reconfigured into three interference-free partitions, two composed of 1 way each and one composed of 2 ways. When and how the partitions are adjusted is determined by a static schedule, generated by a novel real-time scheduling framework.

A crucial contribution of RVMP's interference-free multithreading approach is that it preserves single-task WCET analysis. That is, the WCET of a task can still be derived independent

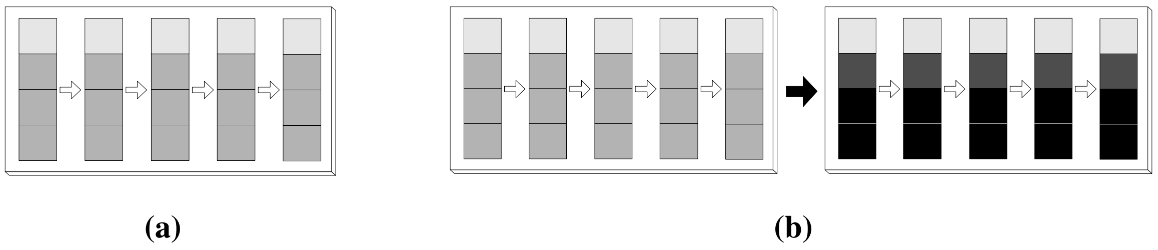


Figure 1-3. (a) Partitioning in space. (b) Rapid reconfiguration in time.

of which other tasks are co-scheduled with it, thanks to interference-free partitions. Single-task WCET analysis is highly desirable because it is tractable (does not need to consider multiple tasks together) and because it provides one constant abstraction of a task (independent of workload context).

Regarding the underlying superscalar processor from which partitions are carved, its complexity is only limited by what WCET analysis tools can handle. Currently, dynamic techniques are beyond the capabilities of most WCET analysis methods. Accordingly, the underlying processor issues instructions in order, and uses static branch prediction and software-managed scratchpad memories (instead of caches). In-order issue does not significantly impact the performance of RVMP. Decoupled virtual processors allow for arbitrary slippage among independent threads, creating an implicit out-of-order execution among different threads. As such, the performance gain from thread-level parallelism offsets the performance loss due to in-order issue within threads. This observation is corroborated by others in the context of in-order SMT processors [55, 44, 66]. Programmatic memory transfers between main memory and on-chip scratchpad memory are often used in real-time applications for determinism [72, 9, 90, 78], not to mention possibly better performance due to programmer/compiler managed layout [91]. It has been shown that static branch prediction actually interacts favorably with WCET analysis, whereas dynamic branch prediction often works against it. Statically predicting the longest path yields a safe WCET, which is also the tightest possible WCET, by virtue of adding the misprediction penalty to what is the shorter path anyway [11].

Note that, the analyzability assumptions for the underlying superscalar substrate (in-order issue, static branch prediction, software-managed scratchpads) are not limitations of RVMP itself. On the contrary, they are artifacts of the capabilities of uniprocessor timing analysis tools, orthogonal to the techniques proposed in this dissertation. For example, if out-of-order (OOO) execution could be handled by WCET analysis, then RVMP would work with OOO execution within VPs.

RVMP is based on the high-performance single-threaded Alpha 21164 4-way in-order superscalar processor [29] as an example starting point. The processor is augmented with replicated register files and program counters to support multiple simultaneous threads, like the Ubicom IP3023 embedded processor [90] or the MIPS 34K embedded processor [78]. Novel pipeline extensions are proposed for aggregating individual ways in both the processor's front-end (fetch, decode, and issue ways) and back-end (multiple heterogeneous execution pipelines), to form one or more interference-free partitions. Forming interference-free partitions is not always as literal as what is physically implied by the high-level examples shown in Figure 1-3. The novelty of our pipeline extensions lies in achieving the effect of physically distinct different-sized partitions, despite the fact that partitions have to share some common processor resources. Novel mechanisms include (described in detail in Chapter 3):

- A custom fetch buffer design facilitates assembling a pre-determined number of instructions for each virtual processor every cycle. This design minimizes impact on the critical instruction fetch unit itself, namely we avoid multiple configurable-width I-scratchpad ports.

- The Alpha 21164 in-order issue stage includes the "slotter" and "scoreboard" logic, responsible for checking data and structural hazards and steering ready instructions to respective execution pipelines. First, we show that the steering datapath is unchanged since all issue slots are connected to all execution pipelines in any case. Second, we identify natural "intervention" points in the control logic for easily decoupling issuing among different virtual processors.

- While the Alpha derivative pipeline fully replicates some function units – *e.g.*, there is a simple integer unit in each of the four execution pipelines – certain function units are only available in some of the execution pipelines (*e.g.*, floating-point units, agen units/D-scratchpad ports). These may need to be shared among multiple virtual processors, seemingly violating the interference-free requirement. This is addressed by conservatively time-multiplexing shared function units, again dictated by the static schedule mentioned earlier. This increases the perceived latency every time the shared resource is used, but worst-case analyzability and overall performance benefits outweigh this localized slowdown in most cases.

- Reconfiguring the number and sizes of partitions often coincides with changing which hardware threads are currently using partitions, again determined by the static schedule. This is not a context switch, just a change in thread selection. However, unlike SMT thread selection, the interference-free requirement stipulates that a deselected thread must appear to instantly relinquish its entire partition before reconfiguring the processor. The processor back-end is not a problem since the execution pipelines are non-blocking – already-issued instructions are allowed to finish execution. Two small shadow buffers (64 bytes each) connected to the blocking decode and issue stages facilitate physically moving the deselected thread's unissued instructions. Moreover, the shadow buffers facilitate moving the preempted instructions back again when the previous configuration is restored. (The new fetch buffer design is inherently non-blocking and does not require a separate shadow buffer.)

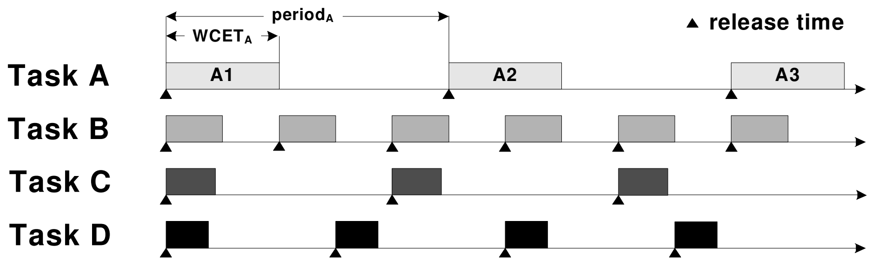
1.2.2 RVMP Scheduling Overview

The processor partitioning and reconfiguration is driven by a static schedule, which is derived by a novel RVMP real-time scheduling framework. The scheduling framework capitalizes on the unique characteristics of the architecture, namely arbitrary interference-free VPs and rapid processor reconfiguration, to produce a simple yet effective scheduling approach that guarantees meeting all deadlines.

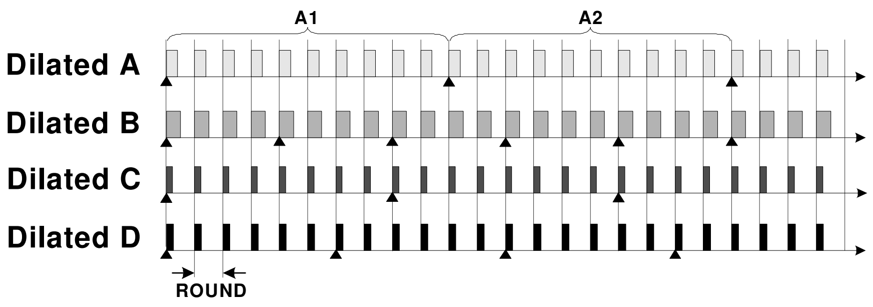
A real-time task-set consists of multiple tasks with different WCETs and periods. For example, Figure 1-4(a) shows a task-set with four tasks. New instances of a task are released based on the task's period, as explicitly highlighted for task A (instances $A1$, $A2$, and $A3$) in Figure 1-4(a). Generating a static schedule (if a feasible one exists) involves considering arbitrary space-sharing and time-sharing of the new architecture among tasks. However, scheduling is impractical for the task-set as shown. Because tasks have different periods, the schedule repeats only after an entire "hyper-period", the least-common-multiple of all tasks' periods [63] (too long to show in Figure 1-4(a)). The hyper-period may be millions of cycles or more, depending on the task-set. Within such a long time span, there is an overwhelming number of possible space/time configuration sequences that must be searched to find a feasible schedule.

Moreover, the dedicated-hardware cost is high, in terms of storing a lengthy static space/time schedule.

RVMP solves this problem by realizing that a hard-real-time task must finish execution just before its deadline, but not necessarily much earlier than the deadline. As such, the execution of each instance of a task can be "spread out" uniformly between its release and deadline without affecting the schedulability of the system as a whole, as shown in Figure 1-4(b). We define a *round* as a small interval of time, say 100 cycles, as in Figure 1-4(b). Each task runs for a guaranteed fraction of the round, called the task's *duty cycle*, then it is temporarily suspended for the remainder of the round, and then it is resumed in the next round during its duty cycle. This process repeats indefinitely, since the completion of a task's dilated instance (*e.g.*,



(a) Undilated tasks.



(b) Dilated tasks.

Figure 1-4. Example task-set.

$A1$ in Figure 1-4(b)) meets with the release of its next instance (*e.g.*, $A2$ in Figure 1-4(b)), intentionally. The dilation process results in an interesting property. The round serves as a "sub-period" common to all tasks. That is, the schedule repeats every round. Thus, a schedule needs to be found for only a *single round*, instead of for the whole hyper-period. Also, notice that this round-based approach makes the exact points of task releases or deadlines irrelevant. All tasks will be available for scheduling during every round, as shown in Figure 1-4(b).

RVMP's round-based scheduling approach considers all possible superscalar way assignments per task, that is 4, 3, 2, and 1 superscalar ways, and finds the duty cycle required by the task in each case to meet its deadline. For example, in Figure 1-5, the duty cycle of task A on 4 ways is depicted at the top-left of the figure. Notice that, naturally, the required duty cycle increases as task A is assigned fewer superscalar ways. However, the WCET of a task (and thus, its required duty cycle) is not linearly related to the number of superscalar ways it is running on (*i.e.*, WCET on two ways is not equal to twice WCET on four ways). This is consistent with the fact that increasing the width of a superscalar processor will lead to diminishing returns of single-thread performance. For a certain assignment of superscalar ways, if the duty cycle duration is greater than the round, then the task will miss its deadline (for example, task B misses its deadline for 1 and 2 superscalar ways as shown in Figure 1-5). Such unsuccessful assignments are discarded. A bin-packing algorithm [*e.g.*,21] is used to try to fit the duty cycles of tasks into one or more architecture configurations within a round, ranging from pure time-sharing (every task uses all 4 ways, sequentially) to pure space-sharing (all tasks run at the same time, one per way). Cases in between pure time-sharing and pure space-sharing involve multiple architecture reconfigurations in the round. Three example failed bin-packing attempts are shown

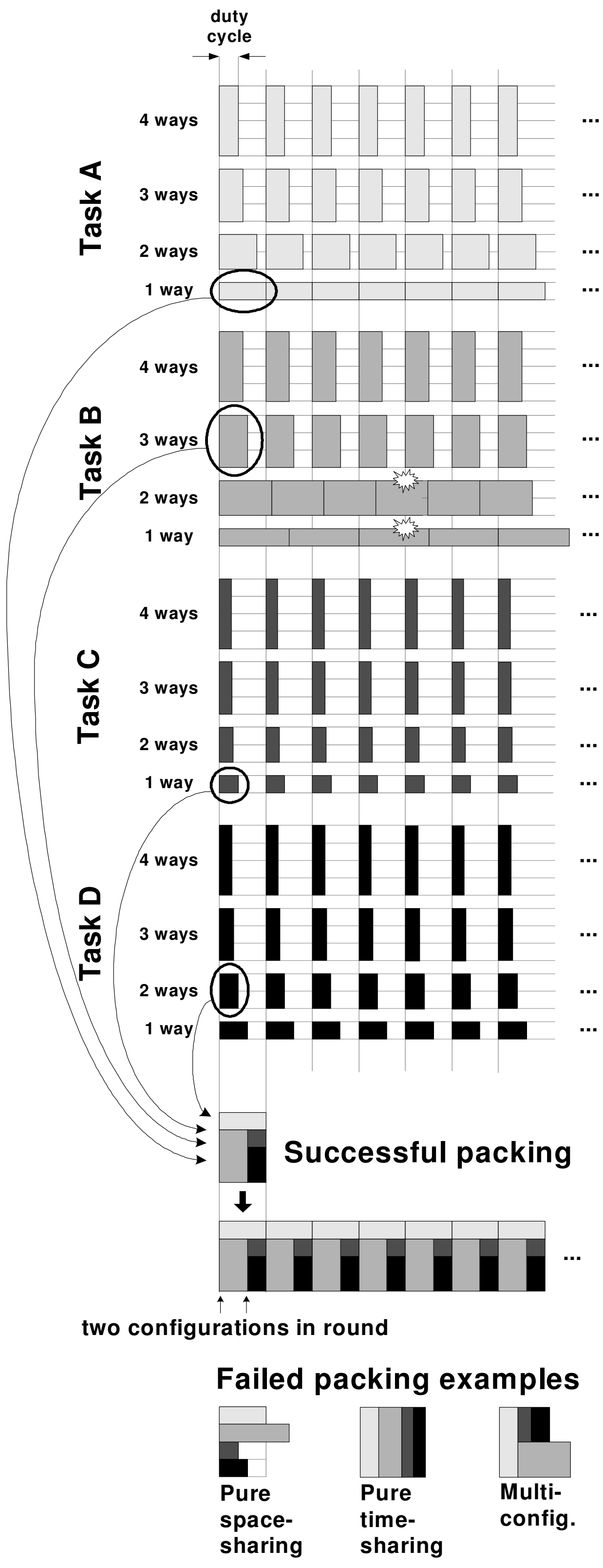


Figure 1-5. Bin packing of dilated tasks.

at the bottom of Figure 1-5, pure space-sharing, pure time-sharing, and a multi-configuration schedule. The successful bin-packing attempt produces a two-configuration round: (1) task A on 1 way and task B on 3 ways, for 60 cycles, followed by (2) task A on 1 way, task C on 1 way, and task D on 2 ways, for 40 cycles. Thus, the processor is reconfigured twice per round. A task-set is considered unschedulable on the architecture if bin-packing fails to find a feasible schedule (in which case the system designer needs to revise task periods, optimize tasks' WCETs, and/or consider higher frequency or different processors, etc.).

The RVMP scheduling approach is simple because it only needs to statically schedule a small time interval, the round. The static schedule repeats indefinitely as shown in Figure 1-5. The efficient static schedule is stored in a compact hardware table that controls the processor partitioning.

A side benefit of RVMP's round-based approach is isolation against rare timing failures. Because each task is guaranteed a certain duty cycle during the round, a single task exceeding its WCET (hence, deadline) cannot overrun the whole system [12].

1.3 Bridging the Processor-Memory Speed Gap via RVMP

The primary advantage of the RVMP framework is that it provides a real-time-compatible multithreading substrate, resulting in analyzable execution overlap among different threads on a superscalar processor. An important side-effect is that RVMP provides a means to hide some of the memory latency of tasks. Although the VP resources of a task become idle when the task initiates a memory transfer, tasks executing on other VPs (owning the remaining superscalar ways of the processor) proceed uninterrupted. As such, the memory access of the task is overlapped with execution of other tasks on other VPs, improving the worst-case performance of the system.

However, the base RVMP framework does not exploit the full potential of multithreading in terms of hiding memory latency. RVMP does not distinguish between pipeline computation and memory transfers of a task, aggregating these two components together into a single overall WCET, just like classic real-time scheduling does. In RVMP, the overall WCET of the task is assumed to be dilated by its duty cycle, as shown in Figure 1-4. However, this is not accurate. Memory transfers do not actually execute on the processor's pipeline: they execute on separate memory transfer units (MTU). As such, a memory transfer is not interrupted by the duty cycle. On the contrary, once a task initiates a memory transfer, it will continue execution on an MTU to completion, whether during the task's duty cycle or not. By not distinguishing between computation and memory and conceptually dilating even the memory component, the base RVMP framework calculates duty cycles that are larger than necessary (*i.e.*, VPs tie up resources longer than needed), missing a readily available opportunity to improve the worst-case utilization even more. We can exploit this opportunity by treating computation and memory as different components of WCET. This is illustrated further by the following example.

Figure 1-6 shows the same task-set of Figure 1-5, running on RVMP with 4 VPs. Let's concentrate on task B, which is running on a 3-way VP, as depicted in Figure 1-6(a). Assume that a new instance of task B is released at the beginning of round i with a deadline (next release) at the end of round $i+2$. RVMP's scheduling framework calculates a safe duty cycle for task B (d_B) that guarantees the task will meet its deadline. Now, assume that task B initiates a single memory transfer (M_B) in round i, to move data from DRAM to on-chip scratchpad

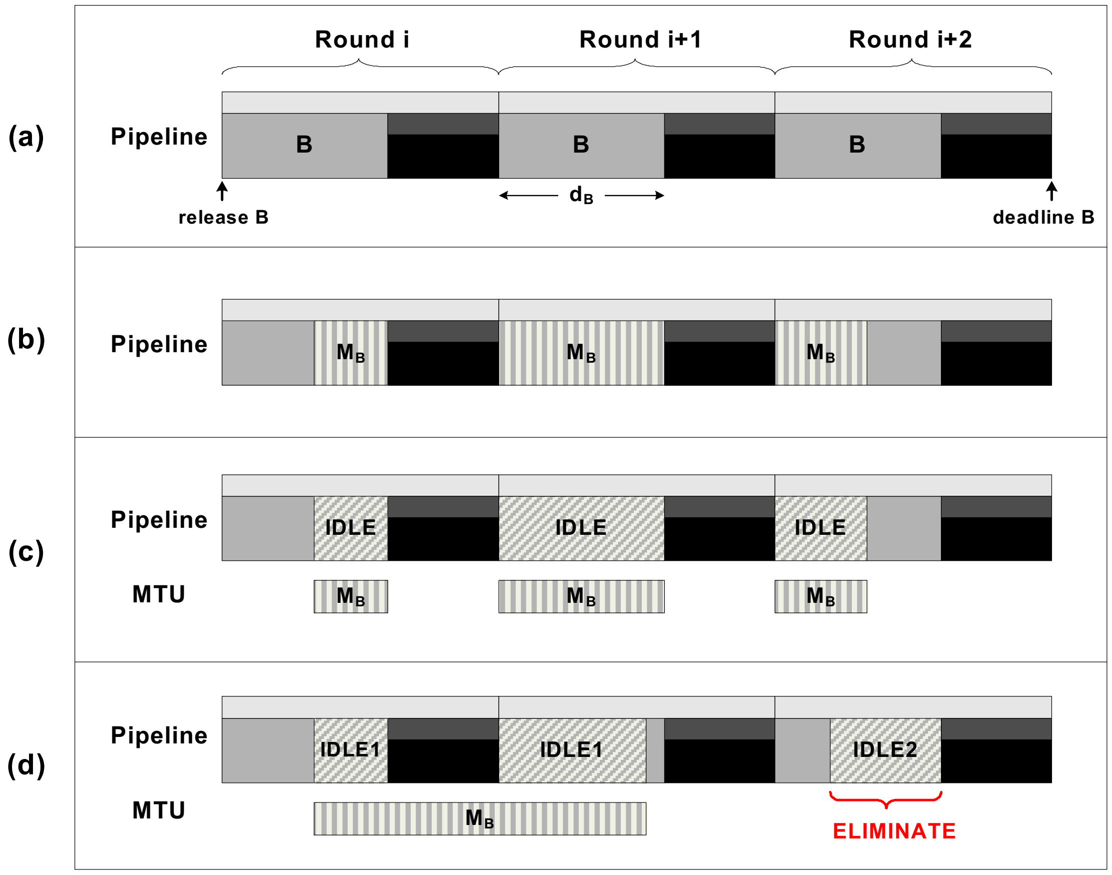


Figure 1-6. Residual resource inefficiency of base RVMP.

(or vice versa), shown in Figure 1-6(b). Since RVMP scheduling does not distinguish between pipeline computation and memory transfers (at least up until now), task B's duty cycle is calculated to provide enough time to accommodate M_B as if it is executing on the pipeline, as shown in Figure 1-6(b). However, in reality, the memory transfer does *not* execute on the pipeline: it executes on a separate dedicated memory transfer unit (MTU). As a result, the pipeline resources assigned to task B's VP will be idle for the duration of the memory transfer. This is shown in Figure 1-6(c): pipeline resources of partition B are idle while M_B is executing on the MTU. Moreover, unlike pipeline computation, the memory transfer is not dilated by the round-based scheduling. On the contrary, it proceeds uninterrupted on the MTU, in the fashion shown in Figure 1-6(d). Now, we can distinguish between two different types of idle time, depicted in Figure 1-6(d):

1. IDLE1: This is the idle time of the pipeline resources (task B's only) during the memory transfer. We need to statically bound this idle time and account for it in schedulability analysis.

2. IDLE2: This is the idle time *after* task B's first instance finishes execution and *before* its deadline. This idle time arises from the fact that we overestimated the necessary duty cycle for running task B, because we did not distinguish between pipeline computation and memory transfers. Ideally, we want to eliminate this kind of idle time by calculating a tighter duty cycle that accounts for the fact that memory transfers proceed uninterrupted (not dilated by duty cycle).

A novel analytical framework can be built on top of RVMP to statically bound the IDLE1 component and eliminate the IDLE2 component, resulting in a safe and tight duty cycle that safely and fully capitalizes on computation/memory overlap to improve the real-time performance of the system and tolerate memory latency.

In essence, the RVMP approach to hiding memory latency is similar to dynamic switch-on-event multithreading [*e.g.*,92,46], where a task is suspended when it initiates a memory access and execution is switched to another task with zero-cycle overhead. However, a key difference between RVMP and dynamic switch-on-event multithreading is that RVMP relinquishes the resources in an analyzable way, whereas switch-on-event multithreading is dynamic, hence, unsafe for hard-real-time systems.

Statically bounding computation/memory overlap is not easy for a multithreading processor that dynamically switches on memory accesses. Determining the schedulability of a task-set requires knowing the exact positioning of memory transfers within tasks *a priori*. Then, all permutations of all tasks in a task-set must be examined for possible overlap opportunities. This exhaustive search may reveal a safe static schedule that can be used afterwords at run-time. This approach is impractical and, most likely, intractable. Instead, a mathematical bound of the computation/memory overlap is desired to avoid searching for a specific overlap opportunity.

The main difficulty arises from the fact that dynamic switching introduces false dependencies among otherwise independent threads. The progress of a given thread is now dependent on other threads initiating memory accesses and relinquishing the pipeline.

This is illustrated in Figure 1-7, which shows two tasks (A and B) running on a scalar switch-on-event multithreading processor. Task A has one memory transfer (M), while task B has two. Each task has access to a private memory transfer unit (MTU), *i.e.*, the system can handle two memory accesses in parallel. In this example, earliest-deadline-first (EDF) [62] is used to prioritize tasks at run-time, giving task A priority over task B (because $deadline_A < deadline_B$). When task A initiates its memory transfer, execution is switched to task B. After the memory transfer of A is over, execution is switched back to A. Note that, the three variants of task A (A, A', and A'') in Figure 1-7 differ only in the position of the memory transfer within the task. As shown in the figure, varying the memory transfer position in task A affects the

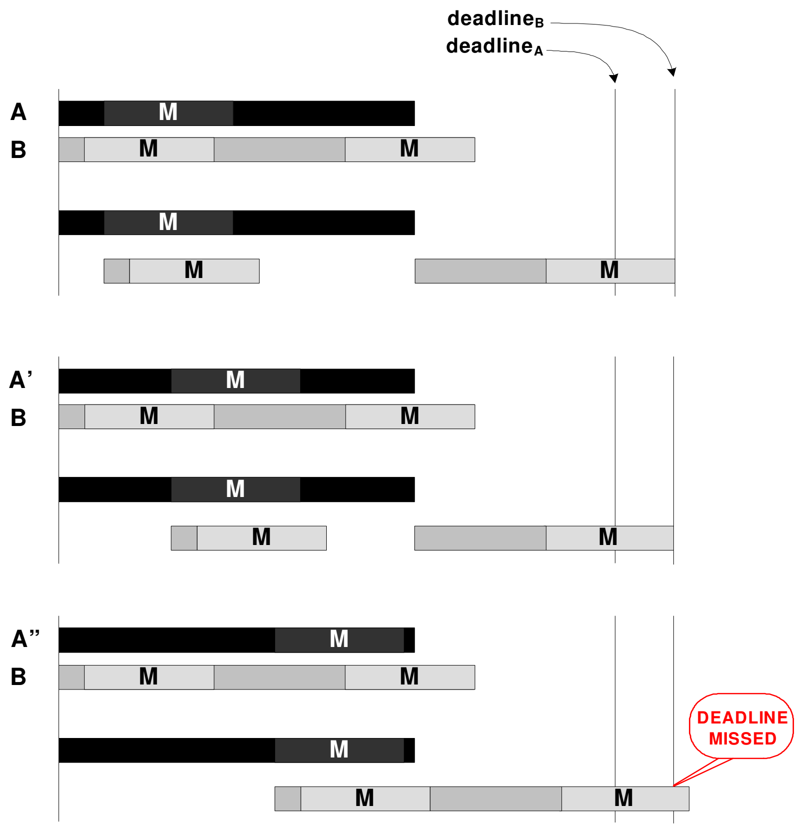


Figure 1-7. False dependencies created by switch-on-event multithreading.

schedulability of task B: task B meets its deadline with task A and task A', but misses its deadline when scheduled with task A''. The problem is not that task B is unschedulable, but rather that the schedulability of task B depends on specific positioning of memory transfers within task A. This implies that schedulability cannot be determined by a simple utilization-based test. Variants of task A – A, A', and A'' – all have the same utilization yet do not yield the same schedulability results. Dynamic switch-on-event multithreading is the reason that a utilization test is insufficient. Instead, schedulability analysis must examine all possible overlap opportunities among tasks to decide if a task-set is schedulable or not. The reason for this complexity is that dynamic switch-on-event multithreading introduces false dependencies between tasks A and B, which are otherwise totally independent.

RVMP's round-based scheduling eliminates this false dependence by switching threads (or VPs) at frequent and regular intervals, totally decoupling threads. A novel analytical framework can be built on top of RVMP to safely and tractably model computation/memory overlap among multiple threads. The frequent and deterministic suspension of tasks during the round forcibly creates memory overlap opportunities: all tasks have equal chances to initiate memory accesses. A task can initiate a memory access only during its duty cycle. By setting the duration of the round equal to the memory access latency, a memory access will finish exactly one round after it is initiated. As such, a precisely determinable fraction of the memory access is overlapped with other tasks' computation. *More crucially, the fraction of overlapped memory does not depend on positioning of memory accesses within and among tasks*.

The concept is illustrated in Figure 1-8, which shows the same two tasks of Figure 1-7 (A and B) running on a scalar RVMP processor (although the concept applies equally to a

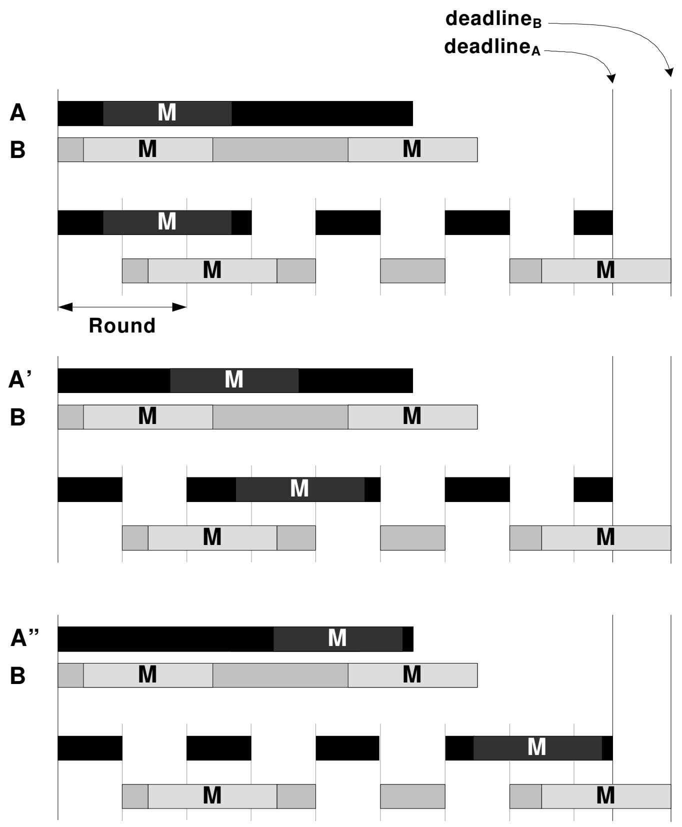


Figure 1-8. RVMP-enabled bounded computation/memory overlap.

superscalar RVMP processor, as described in Chapter 6). Each task runs for 50% of the round before being forcibly interrupted. Notice that, for the three different task A variants (A, A', A'') shown in Figure 1-8, varying the position of the memory access within task A has no effect on the schedulability of task B. Generalizing, tasks are totally decoupled and schedulability does not depend on the positioning of memory accesses within and among tasks. As a result, a tight and safe mathematical bound of computation/memory overlap can be derived knowing only the aggregate computation/memory components of tasks, avoiding an exhaustive search for a particular schedule. Moreover, tighter duty cycles can be calculated taking into account this computation/memory overlap, improving the worst-case performance.

1.4 Contributions

This dissertation makes the following contributions:

1. *RVMP architecture.* Pipeline mechanisms are proposed for virtualizing a single processor into multiple different-sized virtual processors. The virtual processors are truly interference-free despite their creation from a common processor underneath. The architecture combines the analyzability of multiple dedicated processors with the flexibility of SMT. Interference-free virtual processors provide the isolation needed for tractable analysis (both in terms of preserving single-task WCET analysis and facilitating real-time scheduling), thus inheriting the analyzability of multiple dedicated processors. On the other hand, different-sized virtual processors and rapid reconfiguration emulate flexible resource sharing of SMT. Summing up, RVMP is a novel hard-real-time-compatible simultaneous multithreading substrate.

2. *Real-time scheduling framework for the RVMP architecture.* A real-time scheduling framework that interacts closely with the architecture is proposed, yielding a scheduling approach that is both simple and effective. Dilating tasks throughout their periods enables scheduling to be concentrated within a small interval of time, the round. In stark contrast to the alternative of scheduling an entire hyper-period, round-based scheduling is tractable, the round's time span is task-set-independent, and storing the compact static schedule for a round is inexpensive.

3. *Key performance comparisons with rigid MPs and unsafe SMTs*. We demonstrate that schedulability of task-sets is not noticeably affected by excluding out-of-order execution within tasks. Implicit out-of-order execution among tasks on different virtual processors compensates for in-order execution within tasks. This is observed experimentally: RVMP provably schedules task-sets (*i.e.*, for all possible inputs) that pass dynamic testing with *specific inputs* on an unsafe but otherwise equivalent conventional SMT processor. Moreover, RVMP successfully schedules task-sets that are not schedulable on a rigidly-partitioned multiprocessor with equal aggregate resources.

4. *Hierarchical classical/RVMP scheduling*. RVMP naturally supports task-sets which have more tasks than the architecture has virtual processors. This is achieved by assigning multiple tasks to each virtual processor and applying classical uniprocessor scheduling policies to schedule tasks that share a virtual processor (such as earliest-deadline-first or rate monotonic scheduling [62, 63]). The key generalization for multiple tasks per virtual processor: duty cycles are with respect to virtual processors (not tasks) and a virtual processor's duty cycle is based on the combined utilization of tasks sharing the virtual processor.

   As with conventional multiprocessors, determining which tasks to assign to the same virtual processor adds another dimension to scheduling. We first show that conventional assignment approaches extend to RVMP. We then highlight an RVMP-specific dimension, namely that it is beneficial to specifically consider assigning tasks with similar way-preferences to the same virtual processor, since the virtual processor will ultimately receive a single choice of issue width.

5. *Exploiting RVMP to safely tolerate memory latency*. We develop a framework based on the RVMP substrate to safely, tractably, and tightly bound computation/memory overlap among tasks running on different VPs. The framework does not require any knowledge of where memory accesses occur within and among tasks. The only required input is the worst-case number of memory accesses for each task, which is already available as a by-product of the separate and orthogonal WCET analysis phase. A mathematical bound on computation/memory overlap is derived, avoiding an exhaustive examination of overlap scenarios.

   The memory overlap framework integrates seamlessly with RVMP scheduling. The new tightened duty cycles calculated by the memory overlap framework are fed to the RVMP scheduling framework, which is applied unmodified. Moreover, a nice result is that, for the special case of scalar RVMP, the RVMP bin-packing-based scheduling is reduced to a simple closed-form schedulability test, extending the classic EDF utilization test to account for computation/memory overlap.

   The memory overlap framework also accounts for practical memory system issues, such as the degree of parallelism in the memory system (memory banks) and serialization on the bus.

1.5 Organization

The remainder of this dissertation is organized as follows. First, related work is discussed in Chapter 2. The RVMP architecture and scheduling framework are described in detail in Chapters 3 and 4, respectively. RVMP is evaluated in Chapter 5, including comparisons with rigid MPs and unsafe SMTs. The analytical model for bounding memory overlap on top of RVMP is derived in Chapter 6 and evaluated in Chapter 7. The dissertation is summarized in Chapter 8. Finally, Chapter 9 discusses some directions for future research.

Chapter 2

Related Work

2.1 Worst-Case Timing Analysis and Real-Time Scheduling

2.1.1 Worst-Case Timing Analysis

Contemporary static worst-case timing analysis tools can derive tight and safe WCETs of tasks running on in-order scalar [40, 59] and in-order superscalar [61, 64] pipelines. Static branch prediction can be safely accounted for and often yields tighter WCET bounds than dynamic branch prediction [11]. Recent research attempts to derive WCETs of tasks running on out-of-order (OOO) scalar pipelines [58], but so far it is limited by impractical simplifying assumptions, such as oracle branch prediction and artificially small reorder buffers (*e.g.*, 8 entries). Also, it has not yet been proven in the context of superscalar issue. This dissertation provides a more analyzable high-performance alternative. Nonetheless, future techniques for analytically bounding OOO execution can certainly be exploited within our RVMP framework, since

RVMP guarantees non-interference among in-order and OOO VPs alike, a key underlying assumption of WCET analysis (tasks are analyzed individually).

Although a large body of work exists for analyzing the worst-case behavior of caches for real-time systems [*e.g.*,7,52,95,37,42,65,93,76], these techniques are not general enough, especially for data caches, and may produce inflated WCETs. As a result, hard-real-time systems often rely on software-managed scratchpads to achieve the performance of dynamic caches in an analyzable and deterministic fashion [55,44,66].

2.1.2 Uniprocessor Scheduling

A large body of work exists in the area of uniprocessor hard-real-time scheduling [*e.g.*,62, 14, 85, 63]. Hard-real-time scheduling is composed of two components: an off-line (or static) schedulability test and an on-line (or run-time) scheduling algorithm. A schedulability test determines *a priori* whether or not the task-set is schedulable on a target processor under worst-case conditions. Schedulability analysis can either construct an actual schedule to be used at run-time (*e.g.*, cyclic executive scheduling [8]) or just determine schedulability without constructing a schedule (*e.g.*, priority-driven scheduling). At run-time, tasks are scheduled according to the statically constructed schedule (in the case of cyclic executive scheduling) or according to their priorities assigned by the on-line scheduling algorithms. Priority-driven scheduling algorithms can be divided into two categories: static and dynamic priorities [63]. Static priority algorithms, such as rate-monotonic scheduling (RMS) [62], assign fixed priorities to tasks. Task priorities cannot be changed at run-time. In contrast, dynamic priority algorithms assign priorities to tasks at run-time based on certain criteria. For example, earliest-deadline-first (EDF) [62] scheduling assigns highest priority to the task with the nearest deadline, whereas least-laxity-first (LLF) [57] scheduling assigns highest priority to the task with the least available slack time.

The weighted-round-robin (WRR) [63] scheduling policy is fairly simple to implement, similar to cyclic executives. WRR assigns each task a certain percentage or *duty cycle* to run during a round. When the duty cycle is over, the task is interrupted, and the next task in line is scheduled. WRR is used intensively to schedule real-time traffic in high-speed switched networks [*e.g.*,75,48]. However, it perhaps receives less attention in the field of hard-real-time task scheduling because of the prohibitively high context-switching overhead (WRR switches much more frequently than EDF or RMS), an aspect that improves with hardware multithreading support [12]. Our RVMP framework takes advantage of the reduced overhead of context-switching made possible by hardware multithreading to implement an efficient WRR scheduling scheme. Virtual processors are scheduled in a WRR fashion, with each VP assigned enough "weight" to meet the timing demands of its tasks. However, our scheduling framework is unique, because it deals with a superscalar processor that must be shared in both space and time. On the contrary, cyclic executives deal only with scalar processors, which have only time as a shared dimension. In addition, our framework safely, tractably, and tightly accounts for computation/memory overlap to improve performance, which is beyond the capabilities of cyclic executives.

2.1.3 Multiprocessor Scheduling

Real-time scheduling algorithms for multiprocessor systems can be categorized into two groups: partitioning algorithms and global algorithms [26,56]. In partitioning algorithms, a bin-packing approach is used to statically bind tasks to processors [*e.g.*,60], with the goal of load-balancing the processors and/or achieve a feasible schedule for the whole system. Within each processor, a classic uniprocessor scheduling policy is used to schedule tasks. Once a task is assigned to a certain processor, it cannot migrate to a different processor. On the contrary, global algorithms (such as *pfair* [83]) manage all tasks and processors in the system as one dynamic pool. A task can be interrupted and migrated to a different processor in order to dynamically load-balance the processors. In practice, partitioning algorithms are preferred [26] because of their simplicity, effectiveness, and low dynamic overhead. Although global algorithms can better manage tasks and processors at run-time, high overheads and the cost of migrating tasks to different processors often negate their advantage over partitioning algorithms [26,60,83]. In our RVMP framework, we take a partitioning approach to schedule hard-real-time task-sets on multiple virtual processors.

2.2 Multithreading Processors

Hardware multithreading reduces the penalty of context-switching significantly, which facilitates hiding lengthy stalls due to memory accesses and even fine-grain events, such as L1 cache misses, branch mispredictions, and other ILP limiters [80,3,1,87,99,30,89,92]. Moreover, the flexible resource sharing policies made possible by multithreading greatly improve the resource utilization efficiency of the processor, improving the overall performance of the system. However, most prior work focuses on improving average performance, and bounding performance has not been a priority.

2.2.1 Hiding Memory Latency in Real-Time Systems Via Multithreading

Researchers have recently begun exploring coarse-grain switch-on-event multithreading in the context of real-time systems. Switching execution from one task to another at long-latency instructions (*e.g.*, memory) introduces complex scheduling dependencies among otherwise independent tasks, significantly complicating schedulability analysis. Kreuzinger *et al.* [54] provide an empirical study of the effect of overlapping long-latency operations on the schedulability of real-time task-sets. However, they do not provide an analytical framework for provably bounding the amount of overlap among tasks, rather they only perform dynamic testing. Crowley and Baer [24] present a technique to statically bound the combined WCET of tasks which are overlapped via coarse-grain switch-on-event multithreading. They merge the control-flow graphs (CFG) of multiple tasks based on when and where one task can yield execution to another (usually at long-latency events), and find the overall WCET based on the combined CFG. Their technique must consider all possible overlap scenarios among tasks and also it is not compatible with arbitrary scheduling policies. The technique, while safe in terms of worst-case analysis, is limited to scalar pipelines with only one of the hardware threads selected for execution on the pipeline at a time. Our RVMP framework has at least three key advantages. First, RVMP is able to safely exploit multithreading in the context of both scalar and superscalar pipelines. In the latter case, both computation/memory overlap and computation/computation overlap are analytically modeled. RVMP's analytical approach to overlapping tasks' execution on multithreading superscalar processors, as well as hiding memory latency, is a significant performance leap. Second, in the case of computation/memory overlap, RVMP yields a simple analytical approach that does not require exact knowledge of the positioning of memory accesses within and among tasks. Instead, RVMP yields a simple mathematical bound for each task independently, eliminating the need for a complicated analysis of the task-set as a whole. Third, in the case of scalar pipelines, which is the target of Crowley and Baer's technique, RVMP even yields a simple closed-form schedulability test.

2.2.2 Multithreading and Resource Sharing

2.2.2.1 Dynamically Scheduled Superscalar Processors

Several papers [*e.g.*,89, 73, 74, 88, 33, 19] propose and evaluate various policies to share resources among threads in SMT processors, to address both throughput and fairness. However, no hard guarantees can be made regarding the performance of threads, because of the dynamically-scheduled processor underneath and also because no analytical framework is attempted for the general-purpose application space.

Cazorla *et al.* [17,20] further explored SMT resource sharing to provide quality-of-service (QoS) to general-purpose high-priority threads. Performance of a task is continuously checked, and its resource share is adjusted as needed to achieve a certain performance level. Such a dynamic approach is only acceptable in the case of soft-real-time applications, where meeting deadlines is only a matter of QoS. However, for hard-real-time applications, meeting deadlines is a matter of correctness. The authors extended their framework to support one *soft* real-time thread [18]. The real-time thread is assigned just enough processor resources to meet its deadline, and the remaining resources are assigned to non-real-time tasks to achieve high throughput. One major problem with their scheme is that it does not guarantee interference-free multithreading. Even if a certain percentage of processor resources is dedicated to the real-time thread, other active threads will dynamically interfere with it, making it impossible to guarantee the performance of the real-time thread. This means that the real-time thread *can* miss its deadline, even if it is allocated the right amount of resources. Moreover, the amount of resources required for a given performance level is determined by dynamic profiling rather than rigorous worst-case timing analysis, rendering the framework unsafe for critical hard-real-time applications.

Jain *et al.* [49] provide an empirical study of the effects of various SMT resource sharing policies on soft-real-time schedulability. They evaluate various scheduling schemes based on dynamic/static resource sharing and exploiting symbiosis among tasks. Since they target soft-real-time tasks only, they consider a task-set to be schedulable even if some fraction (5%) of deadlines are missed. Schedulability is evaluated on the basis of dynamic testing, which is not suitable for hard-real-time systems. Although they consider statically partitioning resources (only fetch bandwidth and instruction window slots) among tasks to reduce interference, the architecture in its entirety is not safe for hard-real-time applications because it does not guarantee interference-free execution of tasks nor are techniques provided for analytically modeling worst-case interference among tasks.

2.2.2.2 VLIW Processors

Static resource partitioning has been previously proposed in the context of VLIW architectures, such as XIMD [97]. Function units can be grouped to form multiple VLIW clusters with different widths, each cluster with its own sequencer. The use of replicated sequencers and homogeneous function units gives XIMD true multiprocessor qualities, albeit with the ability to gang together FUs to form different-width execution backends. Our work makes it possible to carve out arbitrary interference-free partitions in the context of a contemporary statically-scheduled superscalar processor with one shared fetch unit and heterogeneous function units. Moreover, we also develop a novel real-time scheduling formalism to go with the architecture.

Weld [71, 70] is another multithreading VLIW architecture capable of dynamically combining instructions from different threads into one VLIW word. Threads are part of the same program. A main thread "borks" (branches and forks) a speculative descendant. Instructions from the descendant thread can dynamically utilize resources unused by the main thread. Weld is essentially a speculative multithreading technique [*e.g.*,81,2] that is compatible with a VLIW substrate. The speculative multithreading element is dynamic, tantamount to dynamic scheduling, and is therefore incompatible with analytical frameworks for real-time systems.

A similar dynamic approach for combining instructions from different threads into a single VLIW Multi-OP is used by the M-Machine [51, 36]. Multiple VLIW threads compete for the resources of a single cluster, filling up otherwise idle function unit slots. Again, such a dynamic approach is not suitable for hard-real-time systems. The same applies for other dynamic multithreading VLIW architectures [50, 47].

2.2.3 Miscellaneous Multithreading Processors

The Ubicom IP3023 microprocessor [90] was designed with analyzable high performance in mind. The IP3023 provides 8 hardware threads that share a 10-stage in-order scalar pipeline with static branch prediction and I- and D-scratchpad memories. A 64-entry cyclic table, the hard-real-time table or HRT (a term we borrow in this dissertation), specifies which thread to fetch from next, on a cycle-by-cycle basis. Cycling through threads has certain elements of the HEP [80] and Tera [3] machines. The IP3023 is also quite similar to the scalar DISC architecture [68, 27], which assigns a statically-guaranteed percentage of processor cycles to *execution streams*. The scalar MIPS32 34K [78] is another modern embedded processor that uses a similar approach. The 34K supports 5 hardware threads that can be configured into two virtual processors (VP). A programmable QoS layer determines the percentage of processor resources allocated to each VP, by assigning certain percentages to each thread within a VP (or to a VP as a whole) based on its priority. The 34K also cycles through threads using a 15-entry cyclic table. The IP3023 and MIPS 34K (and precursors, HEP and DISC) are scalar and as such are not concerned with providing interference-free different-width partitions based on aggregating ways of a superscalar substrate. Our work is further unique because we provide a novel analytical framework that bounds computation/memory overlap to safely tolerate memory latency. Thus, our real-time scheduling framework provides a novel means to exploit commercial off-the-shelf multithreading scalar pipelines, such as the IP3023 and MIPS 34K.

2.2.4 Software Thread Integration

Software thread integration (STI) [25,94] is a technique to achieve multithreading performance on low-end microcontrollers without explicit hardware multithreading support. An example application of STI is migrating hardware-implemented functionality into software. Instructions implementing hardware features are inserted into idle slots (slack periods) of a main thread to create a single binary. In typical STI applications, a single non-real-time guest thread is integrated into the fine- and coarse-grain idle periods of a single real-time host thread. Integrating threads is complicated by arbitrarily different control-flow among tasks and/or different task periods, requiring synchronization or padding, or limiting integration opportunities. Our decoupled architecture works with arbitrary real-time task-sets and does not require combined task compilation and analysis.

2.3 Separating Worst-Case Execution Time Components

To perform schedulability analysis, classic real-time theory represents tasks by their abstracted WCETs. Generally, real-time scheduling does not distinguish between different components composing the WCET (*e.g.*, pipeline computation vs. memory accesses), and treats these components uniformly. However, some recent work in the context of real-time dynamic voltage scaling (DVS) [*e.g.*,77, 98, 79] separates the computation and memory components of WCET. Scaling the frequency of the processor scales the computation component of WCET, however, the total time spent accessing memory remains constant and is not affected by frequency scaling. As such, the two components can be separated and treated differently in order to calculate a lower processor frequency than otherwise possible, without jeopardizing the safety of the system. RVMP also separates WCET into computation and memory components, but exploits this separation in a novel context and yields a novel analytical framework accordingly. Since issued memory requests are handled by the memory system and are not pre-emptable, the memory component is not affected by the duty cycle, whereas the computation component is dilated. By realizing this, we can calculate a shorter overall WCET, and as a result, calculate a tighter duty cycle. While the DVS techniques also exploit separation of computation and memory, they do not exploit this separation in the context of multiple tasks to overlap memory of one task with computation of other tasks. Whereas these other techniques address inefficiency in single-task WCET analysis, RVMP exceeds classical schedulability limits of uniprocessors by exploiting intrinsic parallelism in uniprocessors.

Chapter 3

Real-Time Virtual Multiprocessor (RVMP) Microarchitecture

The RVMP processor architecture is built on top of an in-order superscalar processor. The Alpha 21164 [29], an in-order 4-way superscalar processor, serves as a good starting point, partly because of its high-performance emphasis and partly because of available documentation (including an interesting description of its hierarchical issue logic). The RVMP processor architecture is shown in Figure 3-1.

Unlike the single-threaded 21164, RVMP supports 4 thread contexts in hardware, namely 4 program counters and 4 copies of the integer and floating-point register files. Each hardware thread corresponds to one *virtual processor* (VP).

Software-managed instruction scratchpad (I-scratchpad) and data scratchpad (D-scratchpad) memories are used instead of caches for deterministic high-performance. The I-scratchpad is interleaved, having two single-ported banks to guarantee fetching four sequential instructions from a single thread every cycle [*e.g.*,82, 22]. The D-scratchpad has one read port and one read/write port, supporting issuing up to two loads or one load and one store per cycle.

The processor has four integer execution pipelines, $FU0$ (simple integer), $FU1$ (simple integer and integer multiplication/division), $FU2$ (simple integer and load/store address generation), and $FU3$ (simple integer and load/store address generation). There is one floating-point execution pipeline, $FU4$. All function units are pipelined and can accept new instructions every cycle.

Key modifications are made to the fetch and issue stages to achieve the effect of multiple different-sized interference-free VPs, emulating the simplified depiction of Figure 1-3. Light-gray shading in Figure 3-1 highlights the modified fetch and issue stages, discussed in Sections 3.1 and 3.2, respectively.

Dark-gray shading in Figure 3-1 highlights new structures. Two sets of shadow buffers are coupled to the decode and issue stages to support rapid reconfiguration of the processor (Section 3.3). Virtualization is orchestrated by a hard-real-time table (HRT), which contains a static schedule of the processor resources for a single round (Section 3.4).

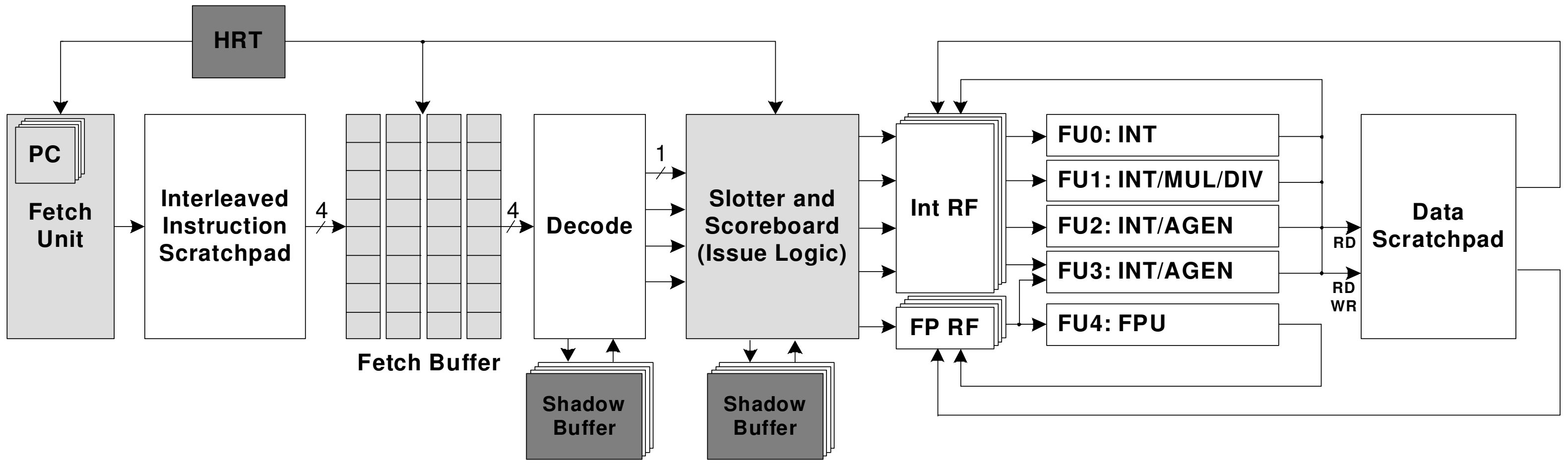


Figure 3-1. RVMP processor architecture.

3.1 Instruction Fetch

Each cycle, the instruction fetch stage must supply a certain number of instructions from each configured (*i.e.*, active) thread, based on the width of the thread's corresponding partition. Moreover, these instructions must be aligned with their corresponding partitions in the subsequent decode and issue stages. For example, for the configuration in Figure 1-3(a), the fetch stage must assemble four instructions every cycle for the decode stage, comprised of one instruction from the 1-way thread followed by three instructions from the 3-way thread. Making the I-scratchpad arbitrarily partitionable requires multiple interference-free configurable-width ports, an expensive prospect in terms of area and complexity. Instead, and to keep the complexity of the fetch unit manageable, we limit fetch to instructions from only one thread per cycle [89, 15, 35]. We then transfer the instruction assembly functionality to a custom instruction fetch buffer, concentrating complexity within a more scalable structure.

The custom fetch buffer serves as a translation mechanism between (1) the single wide fetch port of the I-scratchpad that clearly favors time-sharing and (2) multiple narrower partitions in the space-shared decode/issue stages. A thread's instructions are fetched from the I-scratchpad in individual bursts of (at most) 4 instructions, into a dedicated column of the fetch buffer. Then, the fetch buffer drains and aligns instructions from the column at an even pace that matches the width of the corresponding partition. In more detail:

- In a given cycle, the I-scratchpad delivers (at most) 4 instructions from one thread corresponding to a configured VP. Each VP has a dedicated 8-instruction column in the fetch buffer. Thus, since there are four VPs, there are four columns. The (at most) 4 fetched instructions are written into the corresponding column. Since, in a given cycle, instructions are fetched on behalf of only one VP, the fetch buffer only requires a single 4-instruction-wide write port to write the fetched instructions into the corresponding column.

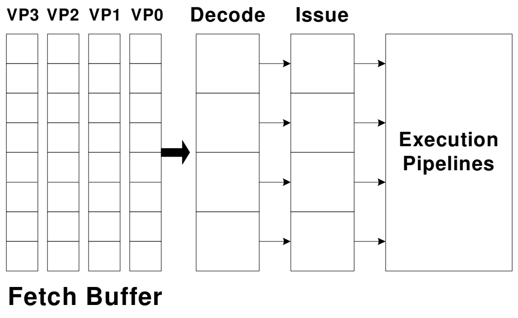
- To maintain equilibrium between incoming (time sharing)/outgoing (space sharing) instructions from each VP's column, VPs are assigned fetch cycles proportional to their partition widths. That is, 1-way, 2-way, 3-way, and 4-way VPs are assigned 25%, 50%, 75%, and 100% of fetch cycles, respectively. Fetch cycle assignment (*i.e.*, which thread to fetch from at any given cycle) is specified by the HRT, as described later in Section 3.4. For example, a 2-way VP inserts (at most) 4 instructions into its column every other cycle. Assuming no stalls in later stages of the VP, 2 instructions are removed from its column – to the decode stage – in each of two consecutive cycles, in time for the next 4-instruction insertion. If a minimum of one cycle is needed between insertion and removal, a column size of 8 instruction slots accommodates all scenarios in which draining is not fully caught up by the time of the next insertion. This definitely is required for a 3-way VP, which inserts 4 instructions in three out of four cycles: consecutive 4-instruction insertions temporarily outrun consecutive 3-instruction removals. This is illustrated by an example later in this section.

- The fetch buffer has four 1-instruction-wide read ports. Each read port can access any instruction in the fetch buffer. Independent fine-grain read ports provides arbitrary configurability, in terms of assembling a certain number of instructions for each configured thread and aligning them with the corresponding partitions in the subsequent decode and issue stages.

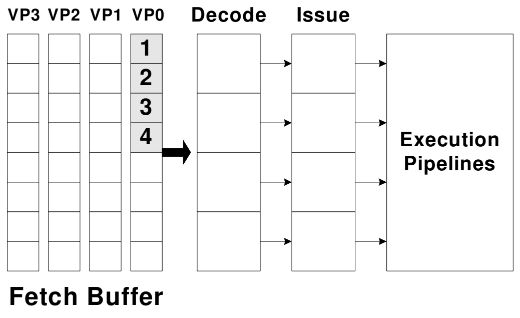
The fetch buffer plays another vital role. When a VP is suspended during the round (due to reconfiguration), its fetched instructions remain in the corresponding fetch buffer column until the VP is resumed during the next round. Per-VP storage in the fetch stage, plus shadow buffers coupled to the decode and issue stages (discussed in Section 3.3), gives each VP the ability to (1) instantly suspend without blocking progress of other VPs and (2) instantly resume from the point it was suspended, preserving the integrity of assumed-suspension-free WCET bounds.

Example. The following example illustrates the operation of the fetch buffer. Assume that the processor is configured into two virtual processors: a 1-way VP_0 and a 3-way VP_1, similar to Figure 1-3(a). Let's also assume, for simplicity, that the processor has a "perfect" pipeline: data and control dependences are resolved instantly, and there are no structural hazards. In other words, the processor will sustain a maximum issue rate of 4 instructions/cycle. The purpose of this example is to confirm that this fetch unit design (time-shared I-scratchpad and custom fetch buffer) is capable of supplying the required instruction bandwidth for each VP to maintain the maximum ideal issue rate of 4 instructions/cycle (*i.e.*, fetch bandwidth is not limiting the performance of the interference-free VPs).

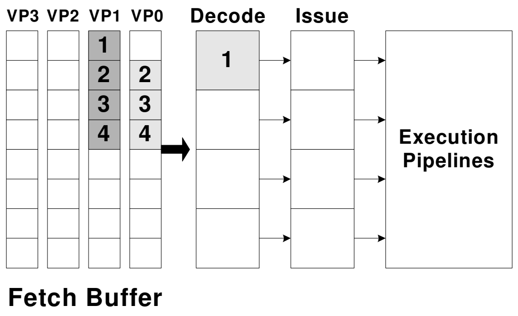
As described earlier, VP_0 (1-way VP) is allocated 25% of the total fetch cycles (1 fetch cycle out of 4), while VP_1 (3-way VP) is allocated 75% (3 fetch cycles out of 4). Initially, the fetch buffer is empty, as shown in Figure 3-2(a). During the first cycle (Figure 3-2(b)), four



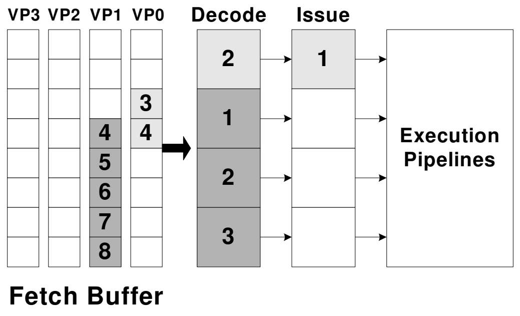
(a) Cycle 0



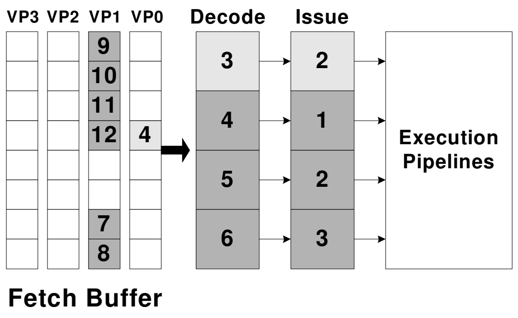
(b) Cycle 1



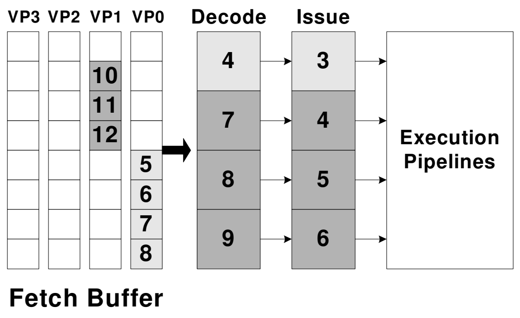
(c) Cycle 2



(d) Cycle 3



(e) Cycle 4



(f) Cycle 5

Figure 3-2. Example of fetch buffer operation.

instructions are fetched for VP_0, and placed in the fetch buffer column corresponding to VP_0. The instructions are numbered (1 to 4) to facilitate tracking them down the pipeline. This is the only fetch cycle for VP_0 during four cycles (including this cycle). The average fetch rate for VP_0 is 1 instruction/cycle, which is the required bandwidth to sustain an issue rate of 1 instruction/cycle for this VP (assuming no stalls).

During the second cycle (Figure 3-2(c)), four instructions (1 to 4) are fetched for VP_1 and placed in its fetch buffer column. At the same time, instruction 1 from VP_0 advances to decode stage (only one instruction because VP_0 is a 1-way VP). In the third cycle (Figure 3-2(d)), four new instructions (5 to 8) are fetched for VP_1. Instruction 1 from VP_0 advances from decode to issue stage, while instruction 2 advances from the fetch buffer to decode stage. Similarly, instructions 1, 2, and 3 from VP_1 advance to decode stage. Notice that, if the size of each fetch buffer column was 4 instructions instead of 8, there would be no space to place all four newly fetched instructions of VP_1 (5 to 8), because instruction 4 is still in the fetch buffer. Only three out of the four fetched instructions could be inserted in the fetch buffer, artificially limiting the instruction bandwidth available for that VP. This is what we meant by instruction fetching outrunning instruction draining.

In the fourth cycle (Figure 3-2(e)), four new instructions (9 to 12) are fetched for VP_1. Instructions from both VPs advance down the pipeline in a similar fashion: instructions in the issue stage are issued to execution pipelines (instruction 1 from VP_0), instructions in the decode stage move to the issue stage (instruction 2 from VP_0, and instructions 1-3 from VP_1), and new instructions are assembled from the fetch buffer and passed to decode stage (instruction 3 from VP_0 and instructions 4-6 from VP_1). This is the third and last fetch cycle of VP_1 during which 12 instructions were fetched, for an average of 3 instructions/cycle. Again, this is the needed instruction bandwidth to sustain an issue rate of 3 instructions/cycle for this 3-way VP.

Over the next four cycles (starting with cycle 5), the same pattern of instruction fetching described above is repeated, starting with VP_0 (four new instructions: 5 to 8 in Figure 3-2(f)), and so on. Clearly, this example shows that the single-ported I-scratchpad, along with the custom designed fetch buffer, is capable of providing the required fetch bandwidth for all VPs while maintaining the interference-free goal of RVMP.

3.2 Instruction Issue

We first briefly explain how instruction issuing works in the single-threaded 21164, as best we can infer from a detailed paper [29]. Then, we discuss key intervention points in the issue logic that are exploited to achieve the effect of interference-free partitions.

3.2.1 Background on 21164 Issue Logic

The 21164 issues instructions strictly in program order. Instruction issue is implemented in two phases, the slot logic and scoreboard logic. The two phases implement hazard resolution hierarchically, first resolving hazards within a fetch/decode group (slot logic) and then resolving hazards between the fetch/decode group and already-issued instructions (scoreboard logic). A high level picture of this logic is shown in Figure 3-3.

The slotter consists of a 4-instruction staging buffer and a routing network (crossbar) for steering instructions from the staging buffer to the execution pipelines. Four instructions received from the decode stage are placed in the staging buffer in program order. Since the 21164 is single-threaded, all four instructions belong to the same thread. The purpose of the first phase is to detect data dependences and conflicts for execution pipelines, only among instructions in the staging buffer. This proceeds in four steps, shown in Figure 3-3:

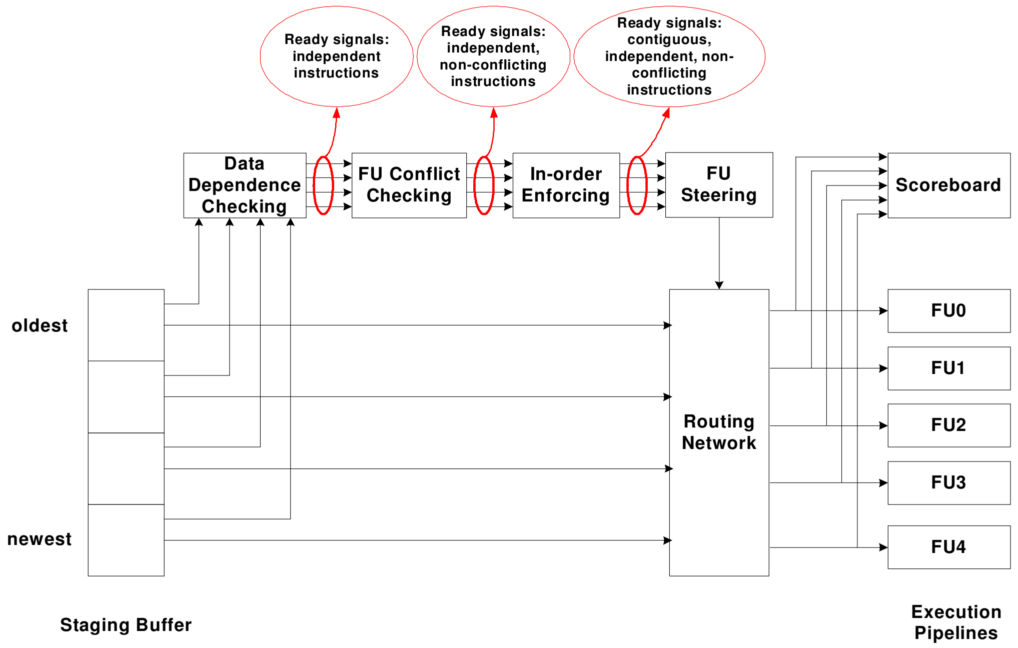

Ready signals: independent instructions
Ready signals: independent, non-conflicting instructions
Ready signals: contiguous, independent, non-conflicting instructions
Data Dependence Checking
FU Conflict Checking
In-order Enforcing
FU Steering
Scoreboard
oldest
newest
Routing Network
FU0
FU1
FU2
FU3
FU4
Staging Buffer
Execution Pipelines

Figure 3-3. Alpha 21164 issue logic.

1. Data dependence checking among staging buffer instructions. This logic compares the source operands of newer instructions in the staging buffer with the destination operands of all older instructions (Figure 3-4). Only data independent instructions are declared as "ready" in this step.

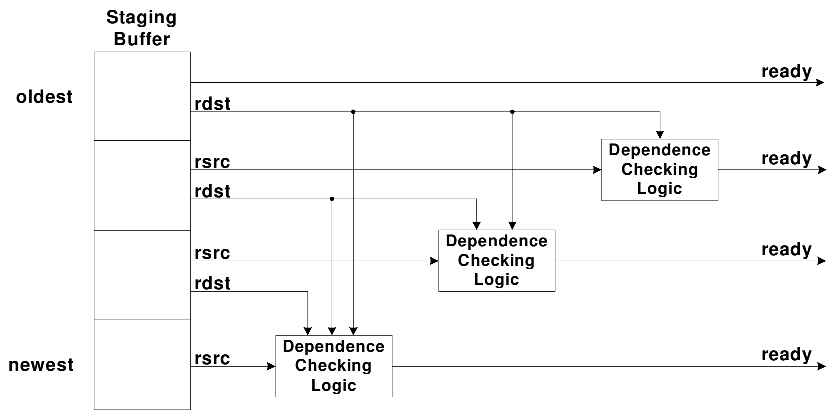

Staging Buffer
oldest
newest
rdst
rsrc
rdst
rsrc
rdst
rsrc
ready
ready
ready
ready
Dependence Checking Logic
Dependence Checking Logic
Dependence Checking Logic

Figure 3-4. Alpha 21164 data dependence checking logic.

2. Function unit conflict checking. This logic arbitrates the execution pipelines among the instructions that were declared "ready" by the previous step. If two instructions are conflicting for the same execution pipeline, only the older instruction in program order is declared as "ready".

3. Enforcing in-order issue. Since the 21164 issues instructions strictly in-order, this logic implements a priority encoding of the ready signals from the previous step: an instruction is declared as ready only if all older instructions in the staging buffer are ready (Figure 3-5).

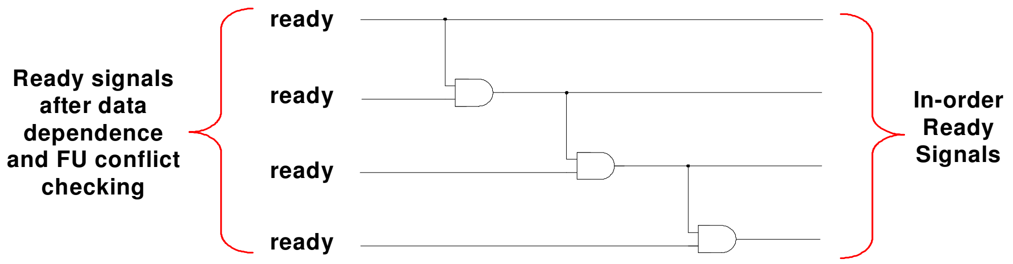

Ready signals after data dependence and FU conflict checking
ready
ready
ready
ready
In-order Ready Signals

Figure 3-5. Alpha 21164: Enforcing in-order issue.

4. Routing network configuration and function unit steering. This logic configures the routing network to steer the ready (i.e., contiguous, independent, and non-conflicting) instructions to their requested execution pipelines.

Instructions that advance from the staging buffer to the execution pipelines check the register scoreboard before issuing, which is the second stage of the issue logic. The scoreboard detects read-after-write and write-after-write hazards between instructions in the issue stage and instructions already in the execution pipelines. When a hazard is detected, the instruction is prevented from issuing to the register read stage and the function unit. Independent instructions that had advanced with it from the staging buffer, which are logically after the instruction in program order, are also stalled from issuing.

3.2.2 RVMP Issue Logic

In RVMP, the datapath associated with the staging buffer does not need to be changed. As before, the crossbar facilitates steering any instruction in the staging buffer to any execution pipeline. An overview of RVMP's issue logic is shown in Figure 3-6. For decoupling issuing among different VPs, we identify four key intervention points in the control logic to work as seamlessly as possible with the existing control logic:

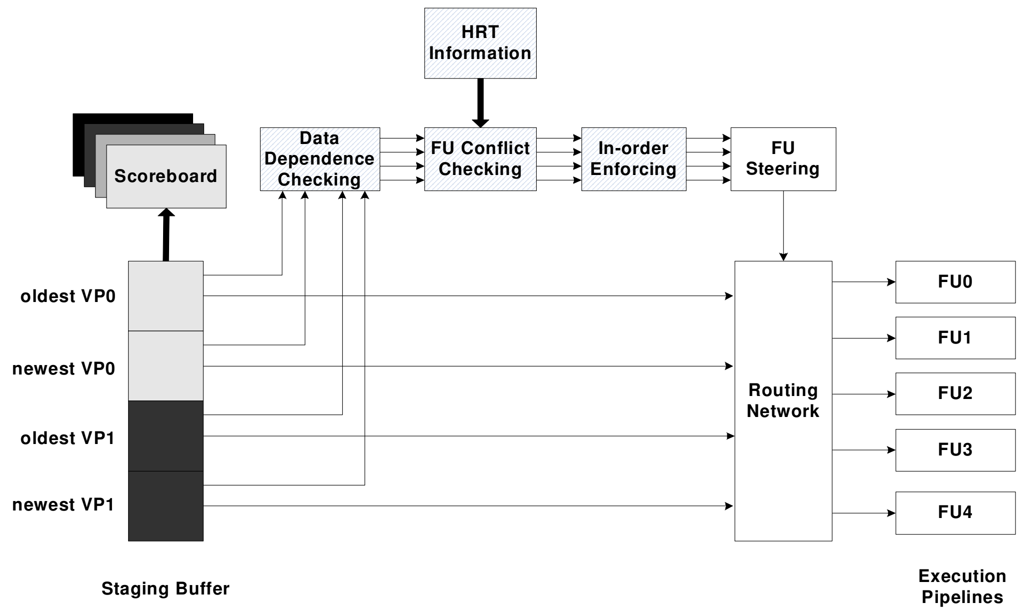


Figure 3-6. RVMP issue logic.

1. The staging buffer may contain instructions from multiple VPs. Fortunately, each VP will have its instructions in contiguous staging buffer entries, in program order, as as-

sembled by the fetch buffer. In Figure 3-4, we highlighted the logic that checks for dependences among instructions in the staging buffer in the 21164. This logic compares the source operands of newer instructions in the staging buffer with the destination operands of all older instructions. Within the staging buffer, the physical arrangement of instructions matches their program order and the dependence checking logic is hardwired accordingly, as shown in Figure 3-4. The match between physical and logical order is preserved within RVMP partitions. This is shown in Figure 3-7, where the RVMP processor is configured as two 2-way partitions. Within each partition, the same order of instructions in the staging buffer is preserved, oldest to newest. Therefore, the hardwired dependence checking logic is compatible with multiple partitions. We only need to include VP IDs in the operand comparisons, depicted in Figure 3-7, thereby partitioning the dependence checking logic among VPs, decoupling the first phase of instruction issuing. An instruction is declared ready only if it is independent of all older instructions within the same VP.

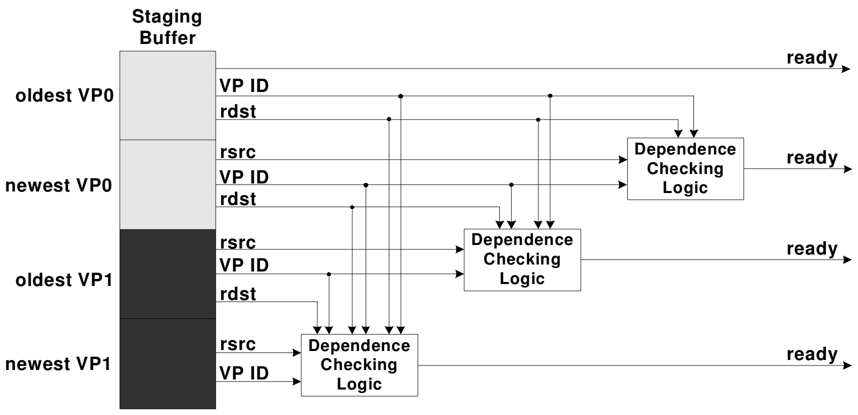


Figure 3-7. RVMP data dependence checking logic.

2. The staging buffer control logic also checks for execution pipeline conflicts among instructions in the staging buffer. In RVMP, conflicts among instructions from different VPs are prevented statically via the HRT (Section 3.4). The HRT specifies the owner (VP) of each execution pipeline every cycle. Thus, the second key intervention point is overriding per-instruction request signals in the staging buffer with ownership information from the HRT. By the same token, conflicts are resolved the same as before for instructions in the same VP: one or more instructions in the same VP may request an execution pipeline if their VP owns it this cycle, initiating arbitration as before. This interaction between the HRT and issue logic is shown in Figure 3-6.

3. Enforcing in-order issue must also be decoupled among different VPs. Instructions from different VPs can issue out-of-order, as long as strict in-order is enforced within a VP. The in-order enforcement logic of the 21164 (Figure 3-5) can be modified to decouple instructions based on their VP ID. A possible implementation is shown in Figure 3-8. The VP ID of each instruction is compared against that of the instruction immediately before it in the staging buffer. If the two VP IDs match, the instruction is declared as ready only if the older instruction is also ready. If the VP IDs do not match, the instruction is declared as ready independently of the previous instruction.

4. Finally, intervention is also needed in the second phase, the scoreboard. Since the multithreaded processor has per-thread register files, it naturally requires per-thread scoreboards (Figure 3-6). Instructions use their VP IDs to lookup the corresponding score-

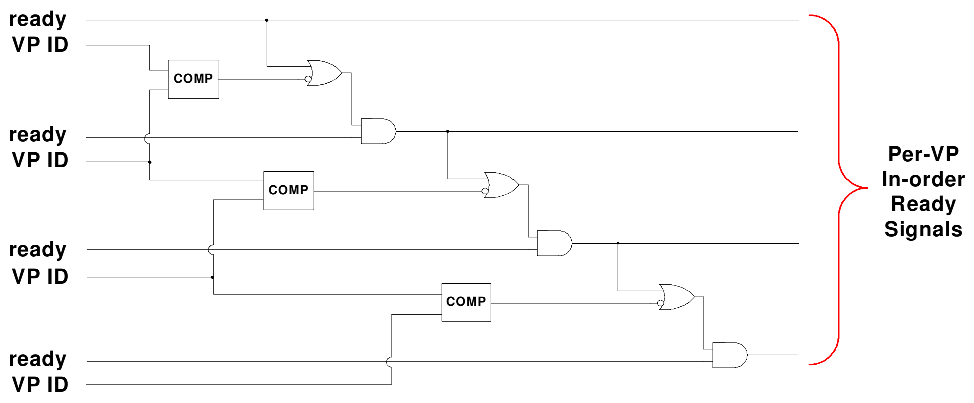


Figure 3-8. RVMP: Enforcing in-order issue within VPs.

board. A stalled instruction only causes other instructions in the same VP to stall. This is achieved by gating stall signals with VP IDs.

There is another subtle difference between the 21164 and RVMP's scoreboarding logic. In RVMP, scoreboard information is checked while the instructions are still in the staging buffer, before advancing to the execution pipelines, as shown in Figure 3-6. If a dynamic hazard is detected, instructions are stalled in the staging buffer entries belonging to their VPs. Execution pipelines should not stall. This is necessary to ensure free-flowing back-end pipelines. Otherwise, if instructions were to stall in the execution pipelines, instructions from other VPs might not be able to proceed in a given cycle even if they own the execution pipeline during that cycle. This violates the interference-free execution paradigm of RVMP.

3.3 Shadow Buffers

Reconfiguring the processor involves suspending one or more of the currently configured VPs and resuming one or more suspended VPs, as depicted in Figure 1-3(c). Tasks' WCETs are derived conventionally, *i.e.*, without knowledge of round-based suspend/resume operations. This means round-based suspend/resume operations must have no perceived execution time overhead (or at least a known worst-case overhead, preferably small, that can be added to tasks' WCETs).

The problem is, at the time of reconfiguration, a newly suspended VP still has instructions in the pipeline. If these instructions are stalled, they will block newly resumed VPs, violating interference-free requirements (single-task WCETs are no longer provably safe). Moreover, instructions of the newly suspended VP will be confused for instructions of one or more newly resumed VPs (as old partitions are repartitioned).

We only need to consider pipeline stages that may block, namely stages in the processor frontend (fetch, decode, and issue). The execution pipelines are free-flowing, so it is safe to allow already-issued instructions of newly suspended VPs to finish. Although the fetch stage is blocking, VPs cannot block each other thanks to dedicated storage per VP in the custom fetch buffer (columns).

We couple a set of shadow buffers to each of the decode and issue stages, to checkpoint/restore the contents of the stages across processor reconfigurations. The number of shadow buffers per set is the same as the number of VPs, since there is a maximum of #VP reconfigurations per round (pure time-sharing). When the configuration of the processor is changed, the four instructions in the decode stage and the four instructions in the issue stage are saved to one of the shadow buffers in each set. During the next round, at the beginning of the same configuration, those instructions are placed again in the corresponding stage latches.

Each set of shadow buffers requires only 64 bytes of storage (4 shadow buffers $\times$ 4 instructions $\times$ 4 bytes).

The shadow buffers are extensions of the pipeline and as such do not pose any unusual problems regarding interrupt handling. If a VP is interrupted, instructions of the corresponding thread must be drained from the pipeline whether or not the VP is currently suspended. The design of any real-time system, conventional or RVMP-based, requires bounding the worst-case interrupt handling latency as well as bounding the worst-case number of interrupts (for example, interrupts caused by classical scheduler invocations in the case of multiple tasks per VP, covered in Section 4.2).

3.4 Hard Real-Time Table (HRT)

The HRT orchestrates the resource sharing among VPs. There are three main facets:

1. *Allocation of fetch bandwidth.* The HRT is responsible for time-sharing the fetch unit among VPs, determining which VP to fetch from each cycle. The HRT thread selection policy guarantees each VP a number of fetch cycles proportional to its superscalar ways.

2. *Partitioning superscalar ways.* The HRT controls how superscalar ways are partitioned among VPs. This is achieved by controlling how instructions are assembled from the fetch buffer before passing them down the pipeline. The HRT determines how many instructions from each VP are assembled each cycle (the number of superscalar ways per VP), and what is the order in which instructions are assembled (which ways belong to which VP).

3. *Allocation of execution pipelines*. The HRT controls the steering logic of the issue stage by providing static "ownership" information for each execution pipeline. To eliminate interference among VPs, an instruction from a certain VP can issue only if its VP "owns" the execution pipeline it requires during the cycle when it's ready. The HRT guarantees each VP function unit bandwidth proportional to its number of ways.

The HRT contains the processor's resource schedule for a single round, as determined by static real-time analysis (Chapter 4). Each entry of the HRT represents a different processor configuration during the round. Since there can be maximum of #VP reconfigurations per round (pure time-sharing case), the maximum number of HRT entries required is also #VP (four in this dissertation). For each configuration (*i.e.*, HRT entry), sharing patterns are encoded using a 4-cycle "mini" schedule for each shared resource. The mini schedules specify the resource bandwidth allocated to each VP, as we explain shortly. Each entry consists of:

- A 12-bit lifetime counter (LTC). The LTC indicates the number of cycles per round for which this entry (configuration) is valid. The sum of the LTCs of all entries equals the duration of the round.
- A 4-entry fetch vector (FV). The FV is a 4-cycle cyclic schedule for the fetch unit, specifying which VP to fetch instructions for in each cycle, for the lifetime of the HRT entry. A 4-cycle schedule is enough to specify the percentage of fetch cycles assigned to each VP based on its number of ways.
- A 4-entry partitioning vector (PV). The PV is used, for the lifetime of the HRT entry, to determine how superscalar ways are partitioned among VPs. The PV controls assembly

of instructions from the fetch buffer corresponding to partitions in the decode and issue stages.

- Five configuration vectors (CVs), one for each function unit. Like the FV, each CV is a 4-cycle cyclic schedule for the function unit. Each entry of a CV indicates which VP owns the function unit during the corresponding cycle.

- A 2-bit cycle count (CC). During the lifetime of an HRT entry, its CC is used to index (i) the FV, to determine which VP owns the fetch unit in the current cycle, and (ii) all five CVs, to lookup which VP owns which function units in the current cycle. The CC is incremented every cycle, wrapping back to zero every fourth cycle. Thus, the 4-cycle sharing patterns specified by the FV and CVs are repeated every four cycles for the lifetime of the entry.

- A 1-bit end-of-table (EOT) flag. The EOT flag is set for the last valid entry of the table.

The HRT is initialized by software before starting a task-set (*e.g.*, system startup). The total required storage space of the HRT is less than 40 bytes. A detailed and generalized equation for calculating the size of HRT is presented in Appendix A.

Initially, a watchdog counter is loaded with the content of the LTC of the first HRT entry (the active entry). The FV, PV, and CVs of that entry are used to configure the processor. The watchdog counter decrements by one each cycle. When the watchdog counter reaches zero, the next HRT entry becomes the active entry, and so on. When the end of the table (EOT flag) is reached, the active entry wraps back to the first HRT entry, corresponding to the beginning of a new round.

Example. Figure 3-9 shows the HRT contents for the example partitioning of Figure 1-5. The static schedule for one round is repeated here for convenience (left-hand side of figure). Recall from that example, there are four tasks in the task-set and each task is mapped to one of the four available VPs. The duration of the round is 100 cycles. There are two different configurations during the round, one active for 60 cycles and the other for 40 cycles. Thus, the HRT contains only two valid entries (the last two entries of the HRT are invalid).

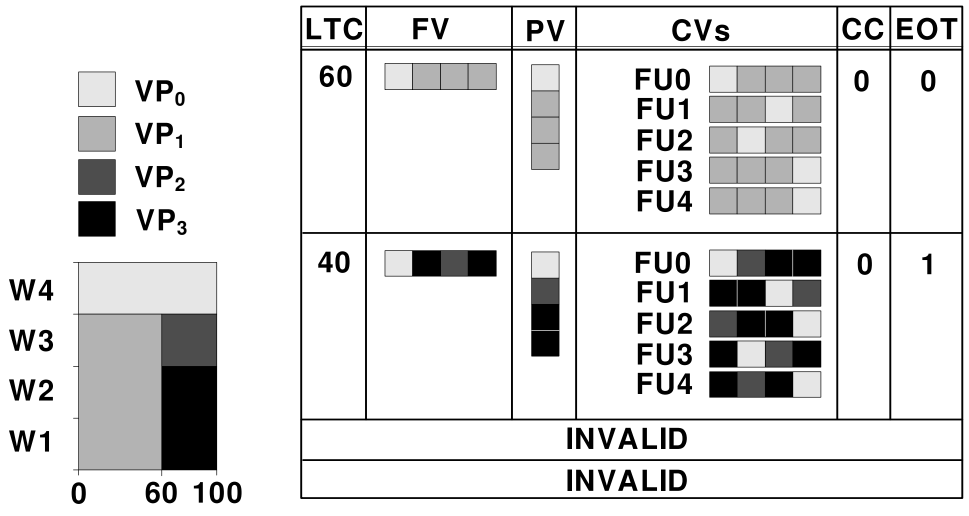


Figure 3-9. Example HRT contents.

The first entry of the HRT indicates that, for 60 cycles, there are two active VPs: VP_0 and VP_1. The superscalar ways are partitioned between the two VPs as indicated by the PV: 1 way for VP_0 and 3 ways for VP_1. The FV determines the nature in which the two VPs time-share the fetch unit: 1 fetch cycle for VP_0 followed by 3 fetch cycles for VP_1, and so on (thus, VP_0 fetches a peak of 4 instructions every 4 cycles, or 1/cycle, and VP_1 fetches a peak of 12 instructions every 4 cycles, or 3/cycle).

The CVs indicate that the 1-way VP_0 owns each function unit for only 1 cycle out of 4 (25% share) and the 3-way VP_1 owns each function unit for 3 cycles out of 4 (75% share). Consider instructions that can execute in any of the four function units $FU0$-$FU3$, namely simple integer instructions. While VP_0 owns each of $FU0$-$FU3$ only 25% of the time, since there are four of them, one of them will be available each cycle for VP_0. Likewise, three of them will be available each cycle for VP_1. Thus, as should be the case, there are no conflicts for simple integer units and no corresponding impact on tasks' WCETs. On the other hand, consider instructions that have a limited number of function units to choose from. For example, integer multiply instructions can only execute on $FU1$. VP_0 owns $FU1$ for one cycle out of four. Thus, WCET analysis safely extends the latency of multiply instructions in VP_0 by three cycles, the worst-case wait time for a ready multiply instruction in VP_0. Similarly, VP_1 owns $FU1$ for three cycles out of four, and WCET analysis safely extends the latency of multiply instructions in VP_1 by one cycle, the worst-case wait time for a ready multiply instruction in VP_1. To sum up, static arbitration for contended units, via the HRT's CVs, makes it possible to bound the worst-case wait time of instructions that use these units.

The second HRT entry in Figure 3-9 indicates that three VPs are active for the remaining 40 cycles of the round: VP_0 (1 way, 25% share), VP_2 (1 way, 25% share), and VP_3 (2 ways, 50% share).

3.5 On-Chip Software-Managed Scratchpads

RVMP employs on-chip software-managed instruction and data scratchpads, similar to those found in Ubicom's IP3023 [90] or MIPS 34K [78]. Software-managed scratchpads provide guaranteed access behavior, eliminating the unanalyzability of the dynamic behavior of caches.

The ISA is augmented with three types of memory transfer instructions: fetch instruction block (retrieve a block from off-chip RAM to the I-scratchpad), fetch data block (retrieve a block from off-chip DRAM to the D-scratchpad), and flush data block (write-back a block from the D-scratchpad to off-chip DRAM). Memory transfer instructions are manually inserted in the tasks (by programmer or compiler) to fetch instruction/data blocks from off-chip DRAM to scratchpads before they are accessed by the instruction fetch unit and loads/stores, ensuring these references always hit. Dirty data blocks that will be re-referenced later are explicitly written back to main memory when they need to be displaced to make room for new blocks.

Since most real-time systems are preemptive by nature, the instruction and data state of a task should be preserved across preemptions. This is usually achieved by partitioning the on-chip memories, and providing each thread with its own private space. This partitioning also eliminates thrashing and conflicts among threads that are active at the same time, greatly simplifying WCET analysis. Cache (or software-managed memory) partitioning is a common technique in real-time systems [53, 96, 67], and is orthogonal to our proposed framework.

Chapter 4

RVMP Static Real-Time Scheduling Analysis

Static real-time scheduling analysis for RVMP is responsible for assigning tasks to VPs and allocating processor resources to VPs (in both space and time) to guarantee that tasks will meet their deadlines. The final output is the HRT contents, corresponding to the space/time schedule of VPs within a round.

We first consider the case in which the number of tasks in the task-set is less than or equal to the number of VPs, thus, only a single task is assigned to each VP (Section 4.1). We then extend the framework to cover the general case of multiple tasks per VP (Section 4.2).

4.1 Single Task per Virtual Processor

The analysis attempts to find the least possible space share (number of ways) and time share for each VP within the round, such that tasks assigned to the VPs will meet their deadlines. Strictly speaking, the analysis does not need to find the most efficient schedule, just the first one that works. Nonetheless, by finding the most efficient schedule, excess resources may be used later to attempt scheduling another periodic hard-real-time task, sporadic (one-time) hard-real-time tasks, periodic soft-real-time tasks, etc.

The analysis proceeds in two steps. First, the analysis produces a space/time schedule for VPs within the round (unless no feasible schedule is found, in which case the task-set is considered unschedulable on the architecture). Second, the contents of the HRT corresponding to the schedule are synthesized.

4.1.1 Generating Space/Time Schedule

Recall that our analysis is based on evenly spreading out the execution of every task over multiple rounds between their releases and deadlines (Figure 1-5). This conveniently enables us to concentrate scheduling within a single round.

Each task is guaranteed a fixed fraction of the round, called a *duty cycle* (d). Since the maximum fraction of time that a task i uses the system *overall* is $U_i = \frac{WCET_i}{period_i}$ (called worst-case utilization), this is naturally the same fraction of the round that task i must be guaranteed. That is, a task's duty cycle is simply its worst-case utilization: $d_i = U_i = \frac{WCET_i}{period_i}$. Since a task's WCET depends on the number of superscalar ways allocated to the VP to which the task is assigned, the task's duty cycle also depends on the way allocation of its assigned VP.

Since the analysis considers a single round and abstracts the processor's resources as superscalar ways, the scheduling algorithm works on a two-dimensional region with area of $R \times W$, where R is the duration of the round (time dimension) and W is the total number of superscalar ways (space dimension). The space/time allotments of VPs are also modeled as two-dimensional regions, each with an area of $(d \times R) \times w$, where w is the number of ways allocated to the VP, d is the duty cycle of the task assigned to the VP assuming w ways, and R is the duration of the round.

The scheduling algorithm considers all possible way allocations (1, 2, 3, or 4 ways) for every VP. For each combination of VPs, we sum the VPs' "areas" (the area of a VP is $(d \times R) \times w$ as explained above). If this sum is greater than the total area available ($R \times W$), the combination is discarded right away because it is impossible to schedule.

Now we need to concentrate on combinations that yield a combined area less than the total available area. For each such combination, we need to fit all the VPs (with their specified superscalar ways and duty cycles) within the overall $R \times W$ region. This is a 2-dimensional bin-packing problem [28]. The bin is a rectangle of width R (duration of round) and height W (total number of superscalar ways). The items we need to pack are the VPs, each of width $d \times R$ (the duration of its duty cycle assuming w ways) and height w (the number of ways allocated to it). Bin packing is an $\mathcal{NP}$-hard problem [28], and there exists a wide range of approximate solutions. Each solution consists of a pre-heuristic, which deals with the order in which the items are packed, and a heuristic, which deals with the packing algorithm itself. The most widely used pre-heuristics are sorting the items (largest first) according to their height, width, area, or perimeter [28]. We use sorting based on perimeter as our pre-heuristic. As for the heuristic, there are plenty of algorithms described in the literature. The *Bottom-Left-Fill* (BLF) algorithm [21] is used here.

The BLF algorithm takes the first item in a sorted list, and finds the bottom- and left-most corner of the bin where the item can fit, and places the item there. This process is repeated until all the items are packed. An example is shown in Figure 4-1. Assume we have four VPs to which tasks A, B, C, and D are assigned, with way allocations as shown. The tasks are characterized as follows (Task(duty cycle,ways)): A(1,1), B(0.6,3), C(0.4,1), and D(0.4,2). Since the pre-heuristic sorts based on perimeter, the sorted list is as follows: B, A, D, C. The BLF algorithm starts with an empty rectangular area of $R \times W$ as in Figure 4-1(a). The first item in the sorted list is task B. The algorithm locates the bottom- and left-most corner, indicated by ($\times$) in Figure 4-1(a), and places B there. The next item is task A, and two corners are located in Figure 4-1(b). Task A will fit only in the upper one, so it is placed there (Figure 4-1(c)). The same procedure is repeated for tasks D (Figure 4-1(d)) and C (Figure 4-1(e)).

Bin packing is repeated for all partitioning combinations (*i.e.*, trying 1, 2, 3, and 4 ways for every VP) that meet the maximum area requirement. Among combinations that succeed the packing algorithm, we select the combination that minimizes the used area (although from

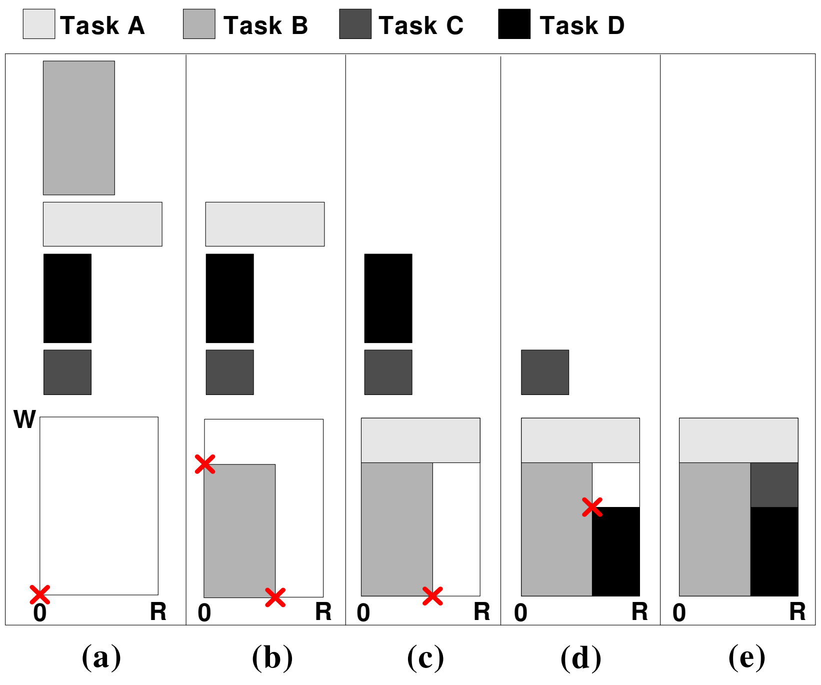


Figure 4-1. BLF packing example.

the standpoint of scheduling only the task-set, the first successful combination would do). The packed schedule of this combination is used to synthesize the contents of the HRT.

4.1.2 Synthesizing HRT Contents

There are only five possible processor configurations given 4 superscalar ways: (a) four 1-way partitions, (b) two 1-way partitions and one 2-way partition, (c) two 2-way partitions, (d) one 1-way partition and one 3-way partition, and (e) one 4-way partition. The HRT entries corresponding to each processor configuration (a)-(e) are manually synthesized and shown in Figure 4-2, respectively. For a given processor configuration, a VP is assigned a fraction of the processor's resources equal to its fraction of superscalar ways. For example, in Figure 4-2(a), each of the 1-way VPs has a 25% share of the processor's resources: each VP owns every function unit (including the fetch unit) for one cycle out of four.

The real-time analysis of Section 4.1.1 may produce a round with multiple processor configurations. For the example in Figure 4-1, Figure 4-2(d) is used for the first entry in the HRT (with an LTC of 60 cycles) and Figure 4-2(b) is used for the second entry in the HRT (with an LTC of 40 cycles and EOT=1).

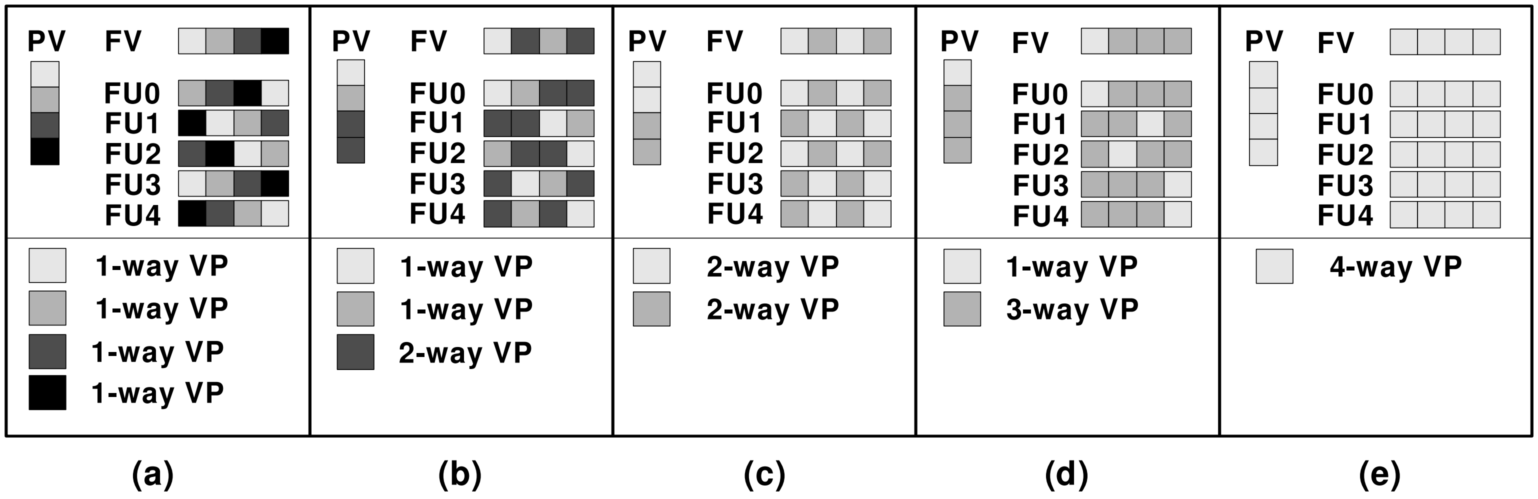


Figure 4-2. The five possible processor configurations for a 4-way superscalar processor.

4.2 Multiple Tasks per Virtual Processor

If there are more tasks in the task-set than the architecture has VPs, then VPs need to support more than just one task each.

The same situation arises in the context of conventional multiprocessors, when there are more tasks than physical processors. In this situation, multiple tasks are assigned to each processor, and classical real-time scheduling policies for uniprocessors – *e.g.* earliest-deadline-first (EDF) or rate-monotonic (RM) [62] – schedule tasks on the same processor. Then, the only question is how to assign tasks to processors, a deeply studied area that is often cast, again, as a bin packing problem with many applicable heuristics [60].

Since VPs are completely decoupled, RVMP can be abstracted as a (flexible) multiprocessor and thus the same techniques apply.

1. Multiple tasks per VP are naturally accommodated by applying (for example) EDF scheduling within VPs. A VP's duty cycle must now accommodate the combined utilization of all tasks assigned to it. That is, the duty cycle of a VP is simply the sum of the duty cycles of tasks assigned to it. Since our procedure of Section 4.1 bin-packs VPs, not tasks, the procedure transparently extends to multiple tasks per VP as long as VPs' duty cycles are calculated using the generalization above.

2. We apply bin packing once again to ensure a good assignment of tasks to VPs, with a subtle enhancement. Since tasks sharing the same VP will ultimately execute on the same number of superscalar ways (the number of ways ultimately allocated to a VP does

not vary) an additional consideration when grouping tasks is whether or not they have similar (in-order) instruction-level parallelism (ILP).

Chapter 5

Evaluation of RVMP Framework

5.1 Experimental Methodology

The primary experiments involve static worst-case schedulability analysis, which determines the ability to schedule task-sets on various architectures. We compare worst-case schedulability of task-sets on the proposed RVMP architecture and several conventional multiprocessors with equal aggregate resources. Since there is no known static worst-case schedulability analysis framework for SMT (unsafe), it is excluded from the primary experiments.

The primary experiments are followed by secondary experiments, proof-of-concept simulations of the various architectures. Note, dynamic testing does not prove the schedulability of a task-set in the worst-case, only its schedulability for the particular task-set inputs used. Detailed microarchitectural simulation is useful as a prototyping exercise (proof-of-concept) and it also provides a medium for comparing run-time performance of RVMP and SMT for particular task-set inputs.

For the secondary experiments, we use a custom detailed cycle-level simulator that faithfully models the RVMP architecture described in Chapter 3. The custom simulator is based on the SimpleScalar toolset, supporting the SimpleScalar ISA (PISA) [13]. The simulator also models conventional multiprocessor and SMT architectures. Conventional SMT uses out-of-order execution (64-entry reorder buffer), dynamic branch prediction (*gshare* predictor with 2^{16} entries), and hardware-managed caches (the same sizes as RVMP's software-managed scratchpads, borrowed from Ubicom's IP3023 microprocessor [90]: 256KB I-scratchpad and 64KB D-scratchpad). The microarchitecture parameters are summarized in Table 5-1.

All task-sets are simulated for a complete hyper-period or 100 ms, whichever is less. In these secondary experiments, we compare the run-time performance of (1) RVMP, (2) various equivalent multiprocessor systems with equal aggregate resources, and (3) (unsafe) SMT. Run-time performance is compared in terms of successfully meeting all deadlines or not.

Notice that, the experiments in this Chapter assume a somewhat idealistic memory system. Contention for memory bus bandwidth and conflicts for DRAM banks are not modeled in these experiments. A detailed model for safely accounting for memory contention issues is presented in Chapter 6 and evaluated in Chapter 7. The results presented here are still valid and representative, because all the simulated models have a common baseline of equal advantage over the more realistic memory model of Chapter 6.

Static worst-case timing analysis (*i.e.*, deriving tasks' WCETs) is briefly covered in Section 5.1.1. We then characterize the tasks and task-sets used in our experiments, in Section 5.1.2.

Table 5-1. Microarchitecture parameters.

Processor Core	Simplescalar PISA ISA
	Processor frequency = 1 GHz
	RVMP Microarchitecture (Figure 3-1) 4-way in-order superscalar 4 thread contexts 4 entry HRT Static (BT/FNT) branch predictor
	SMT models out-of-order core with 64 entry ROB Gshare predictor (2^{16} entries)
	Function units: 1 INT, 1 INT/MUL/DIV, 2 INT/AGEN, 1 FPU, 4 MTU
Core Latencies	Address generation = 1 cycle Integer ALU ops = 1 cycle Complex ops = MIPS R10K latencies
Level-1 Scratchpad Memories (on-chip)	*based on Ubicom IP3023* Instruction scratchpad: 256KB, interleaved Data scratchpad: 64KB, 1RD & 1RD/WR ports (dynamic caches for SMT models) Block size: 128 bytes Access time: 2 cycles
Memory System	Bus frequency = 500 MHz Bus width = 4 bytes DRAM banks = 4 DRAM access time = 50 ns/block (no conflict) Bus transfer time = 64 ns/block (no contention)

5.1.1 Static Worst-Case Timing Analysis

The real-time scheduling analysis presented in Chapter 4 requires the WCET for each task, for each of 1-way, 2-way, 3-way, and 4-way partitions. These are referred to as $WCET_1$, $WCET_2$, $WCET_3$, and $WCET_4$, respectively. We cannot simply assume that $WCET_4 = \frac{1}{4} \times WCET_1$, because performance does not scale linearly with the number of superscalar ways. Moreover, the worst-case-extended instruction latencies caused by time-sharing contended function units (as explained previously in Section 3.4) are different among the four cases. WCET analysis needs to be performed specifically for each partition width.

Although we have access to static worst-case timing analysis tools capable of bounding WCETs of hard-real-time tasks on simple pipelines, it is beyond the scope of this dissertation (and orthogonal to it) to port one of these tools to model the RVMP microarchitecture. Thus, we perform manual analysis assisted with simulation to safely yet tightly bound tasks' WCETs.

Our WCET analysis is procedurally similar to the bottom-up fixed-point approach described by others [43]. WCET analysis involves identifying the longest timing paths in the program, moving upwards from inner loops and leaf functions towards outer loops and functions at higher levels. Forward branches are handled by selecting the longer of two timing paths, after padding the taken path with the misprediction penalty, since our static branch prediction heuristic predicts forward branches as always not-taken. Backward branches are handled by padding the loop continuation with the misprediction penalty, since our static branch prediction heuristic predicts backward branches as always taken. After identifying longest timing paths, we use simulation to tightly model overlapped execution of instructions along these paths. Note that the I- and D-scratchpads are partitioned among tasks to eliminate interference and improve analyzability, a common practice in hard-real-time systems [53, 96].

For RVMP (or multiprocessor) cases where more than one task runs on the same VP (or processor), tasks are scheduled using EDF within a VP (processor). The overhead of the scheduler itself must be accounted for in the WCET. For each task, the worst number of scheduler invocation is two: one when the task is released, and one when it completes [5]. Thus, the WCET of each task is padded with twice the worst-case execution time of the scheduler.

5.1.2 Real-Time Tasks and Task-sets

We use benchmarks from the C-lab real-time benchmark suite [16] and MiBench embedded benchmark suite [38], shown in Table 5-2. These benchmarks are compiled to the SimpleScalar PISA ISA [13] with -O3 optimization enabled. The first column in Table 5-2 shows the benchmark names. The second through fifth columns show four WCETs for each task, for each of 1, 2, 3, and 4 ways, respectively.

Table 5-2. Benchmarks (WCETs in ms at 1GHz processor).

Task	$WCET_1$	$WCET_2$	$WCET_3$	$WCET_4$
CNT	0.118	0.0929	0.0777	0.0777
ADPCM	3.06	2.29	1.86	1.64
SRT	2.26	1.51	1.13	1.01
MM	2.93	2.29	1.97	1.97
FFT	0.692	0.526	0.496	0.447
LMS	0.205	0.140	0.123	0.0963
CRC	0.0594	0.0513	0.0434	0.0434
TOAST	0.347	0.261	0.253	0.231
LAME	9.79	7.64	6.95	6.27

Using the tasks above, we generate numerous task-sets with 4 tasks and others with 8 tasks. Tasks are randomly selected for each task-set. The period of every task is randomly selected such that $WCET_4 \leq period < 4 \times WCET_1$ (or $8 \times WCET_1$ for task-sets with 8 tasks). The lower bound on the period ensures that any single task will at least be schedulable on a 4-way in-order processor. The upper bound on the period provides some slack for the task-set as a whole to be possibly schedulable.

We define the *scalar utilization* (U_{scalar}) of a task-set as the sum of its tasks' worst-case utilizations according to their 1-way WCETs ($\sum_{\tau} \frac{WCET_{\tau,1}}{period_{\tau}}$, for all tasks τ in the task-set). Task-sets are sorted into four different categories (or bins) based on their U_{scalar}. The four bins are as follows: $0 < U_{scalar} \leq 1$, $1 < U_{scalar} \leq 2$, $2 < U_{scalar} \leq 3$, and $3 < U_{scalar} \leq 4$. Each bin has 25 randomly-generated task-sets. These bins represent increasing difficulty in scheduling a task-set, the first bin containing the least demanding task-sets and the fourth bin containing the most demanding task-sets. Any task-set with $U_{scalar} > 4$ is provably unschedulable on all architectures used in the primary experiments.

5.2 Results

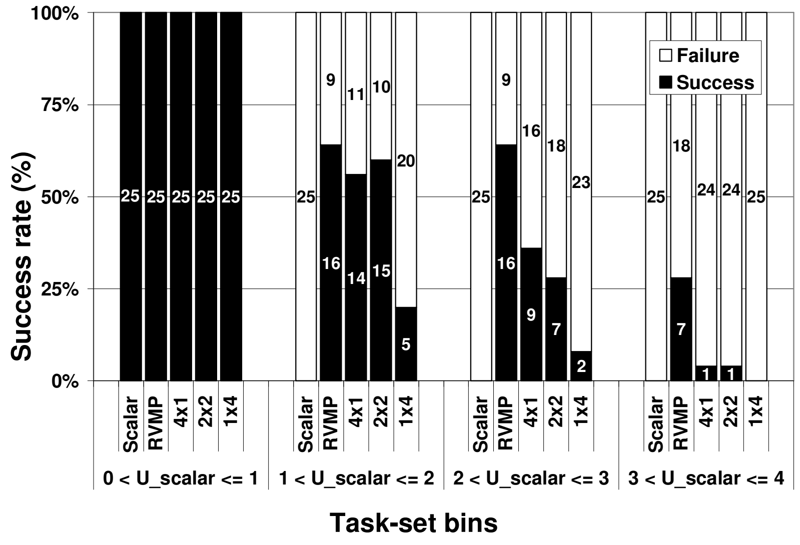
5.2.1 Schedulability Tests

The graph in Figure 5-1 shows worst-case schedulability results, for various statically analyzable architectures. Figure 5-1(a) is for task-sets with 4 tasks each and Figure 5-1(b) is for task-sets with 8 tasks each. For each utilization bin, we plot the number of task-sets in that bin that are schedulable ("Success") versus not schedulable ("Failure") in the worst case, for the various architectures. "Scalar" is the in-order scalar processor, which we used to calculate the scalar utilization (U_{scalar}) for each of the task-set bins. “RVMP” is our proposed real-time virtual multiprocessor. The other three bars correspond to classic earliest-deadline-first (EDF) scheduling on various conventional uniprocessor and multiprocessor configurations: “4$\times$1” (four in-order scalar processors), “2$\times$2” (two in-order 2-way superscalar processors), and “1$\times$4” (a single in-order 4-way superscalar processor). All architectures have the same frequency (1 GHz). All architectures (except “Scalar”) have equal aggregate resources (equal aggregate fetch, issue, and function unit bandwidth). For the conventional multiprocessor systems (“4$\times$1” and “2$\times$2”), the *first-fit-decreasing-utilization* algorithm [69] is used to assign tasks to processors when there are more tasks than processors.

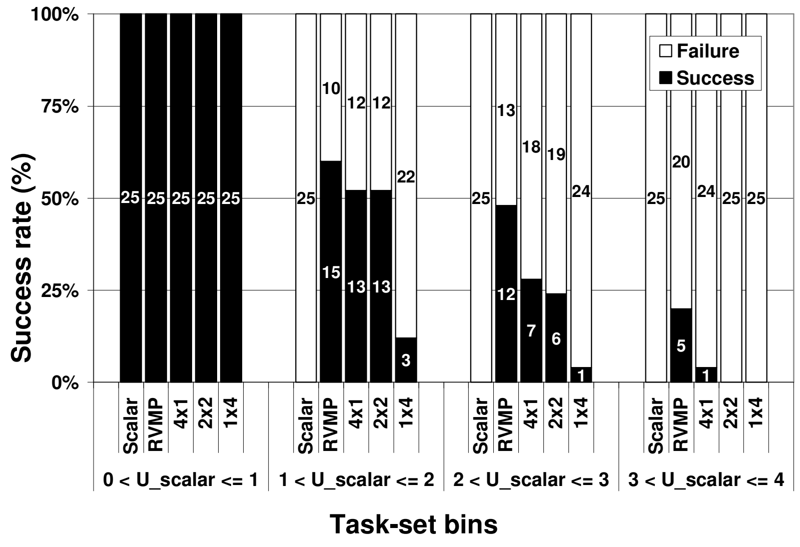
The numbers on the bars represent how many task-sets succeeded and how many failed (out of a total of 25 task-sets per bin). For example, for the second utilization bin ($1 < U_{scalar} \leq 2$) in Figure 5-1(a), 16 task-sets are schedulable on “RVMP” and 9 task-sets are not, in the worst case.

Task-sets in the first utilization bin are provably schedulable on an in-order scalar processor because their scalar utilizations are less than or equal to 1, therefore we expect these task-sets to be schedulable on all five architectures. This is confirmed in Figure 5-1(a) and Figure 5-1(b): all 25 task-sets are schedulable (“Success”) on all five architectures. On the other hand, task-sets in the three higher utilization bins ($U_{scalar} > 1$) are provably unschedulable on “Scalar”, as confirmed in Figure 5-1.

As we move from lower to higher utilization bins, scheduling task-sets naturally becomes harder. In all cases, however, “RVMP” successfully schedules more task-sets than all the other architectures, demonstrating greater flexibility compared to conventional rigid multiprocessors.



(a) 4 tasks per task-set



(b) 8 tasks per task-set

Figure 5-1. Worst-case schedulability analysis.

Moreover, flexibility becomes more important for more demanding task-sets. For example, from Figure 5-1(a), “RVMP” schedules 7 task-sets in the highest utilization bin, whereas the next best architectures (“4$\times$1” and “2$\times$2”) schedule only 1 task-set.

The single in-order 4-way superscalar processor, “1×4”, successfully schedules considerably fewer task-sets than the other architectures, across the board. This is due to the lack of out-of-order execution of either kind: no OOO execution within tasks (necessary for analyzability) and no OOO execution among tasks (“1×4” is single-threaded).

Figure 5-1(b) shows that “RVMP” is scalable in terms of supporting more tasks than VPs. For task-sets with 8 tasks, two tasks are scheduled on each VP using classic EDF scheduling within each VP.

Figure 5-2 shows a histogram of the various processor configurations used by “RVMP” to schedule the task-sets of Figure 5-1(a). (For example, “1-3/2-2” denotes two configurations in the round: (i) a 1-way VP and 3-way VP, and (ii) two 2-way VPs.) For task-sets with $U_{scalar} \leq 1$ (not shown here), “RVMP” was configured as four 1-way partitions with no reconfigurations during the round. This is expected due to the low demand of task-sets in that bin (all task-sets were successfully schedulable on the scalar “Scalar”). Notice however, that “RVMP” shifts more and more to flexible configurations as the task-sets become more demanding (higher U_{scalar}). This observation is consistent with the results of Figure 5-1(a). In the highest bin, “RVMP” is still able to schedule 7 demanding task-sets whereas “4×1” and “2×2” only schedule 1 task-set each. Flexible configurations are clearly valuable in this regime.

5.2.2 Run-Time Experiments

In Figure 5-3, we show the number of task-sets that succeed or fail at run-time on the various architectures, using our cycle-level simulator. A task-set is considered successful if all deadlines are met for the simulated time-frame (the lesser of the hyper-period or 100 ms). To

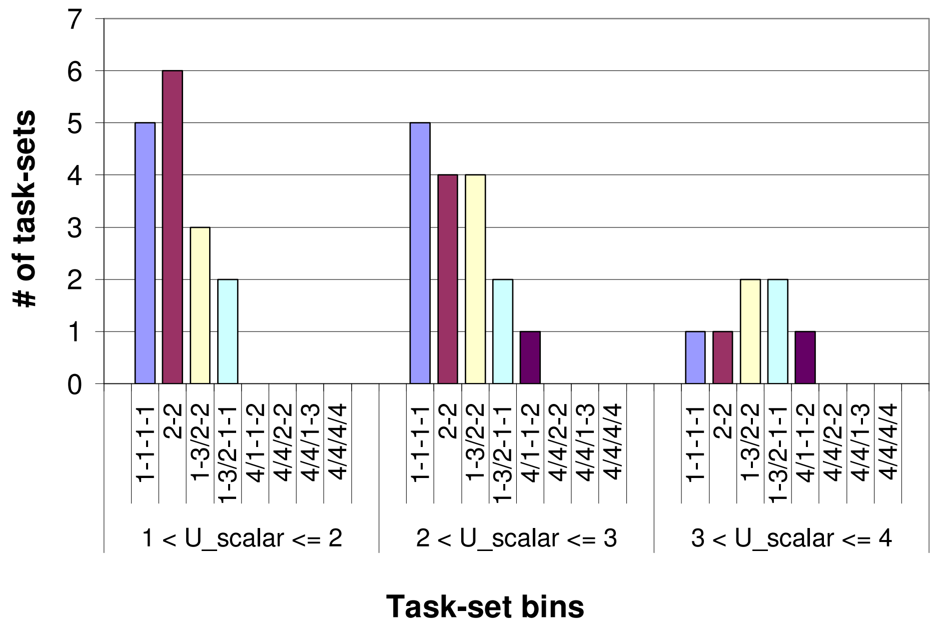
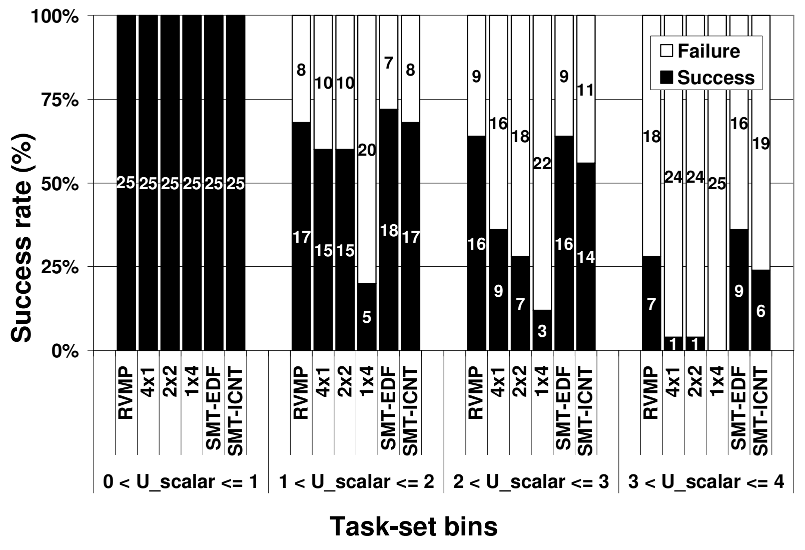


Figure 5-2. Configuration histogram.

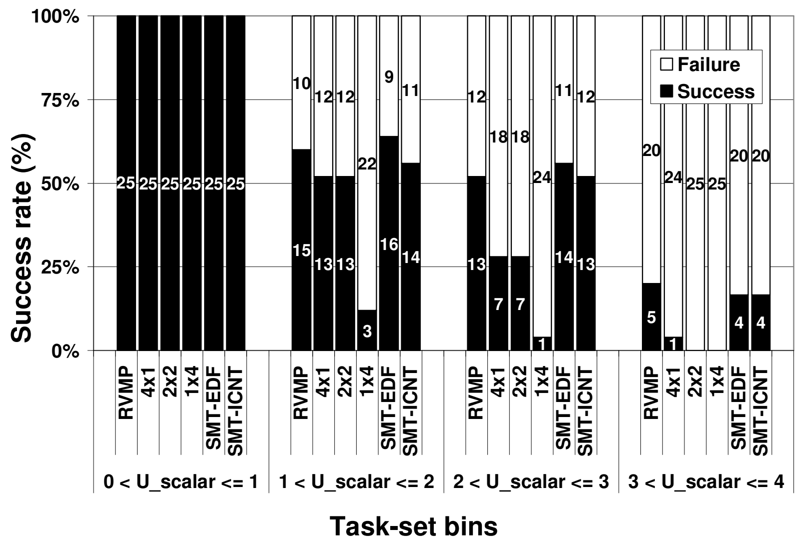
reiterate, whereas *formal* schedulability results from the previous subsection hold in the *worst case*, regardless of the task-set inputs, simulation results in this section only hold for *specific* task-set inputs.

Two unsafe SMT architectures are now introduced, in addition to the safe architectures used previously for formal schedulability tests: “SMT-EDF” and “SMT-ICNT”. “SMT-EDF” uses a real-time-aware (but still not provably safe) thread selection policy that prioritizes tasks according to earliest deadlines [6], while “SMT-ICNT” uses the classic ICOUNT [89] thread selection policy.

The run-time results for the statically analyzable architectures are in agreement with our schedulability tests of the previous subsection. The slight differences between Figure 5-1 and Figure 5-3 are due to differences between WCET estimates and actual execution times: actual execution times with specific task-set inputs can naturally be less than WCETs. For example,



(a) 4 tasks per task-set



(b) 8 tasks per task-set

Figure 5-3. Run-time experiments.

for the second utilization bin in Figure 5-1(a), "RVMP" successfully schedules 16 out of the 25 task-sets. However, according to Figure 5-3(a), 1 additional task-set – only for specific inputs to the task-set – is successfully scheduled at run-time.

Not only is “RVMP” compatible with hard-real-time system design from the standpoint of a formal schedulability framework, but it also performs comparably to the two dynamic SMT architectures in the run-time comparison. For specific task-set inputs, “SMT-EDF” successfully schedules at run-time no more than two extra task-sets over what “RVMP” schedules. “SMT-ICNT” never successfully schedules more task-sets at run-time than “RVMP” (although they are also close).

These results indicate that, although “RVMP” is more coarse-grain in its space/time partitioning than SMT, the partitioning is flexible enough in practice to match SMT, thus successfully combining both analyzability and high performance. With dynamic SMT, on the other hand, there is no way to tell *a priori* which task-sets will succeed in the worst case. As such, it is unsafe to rely on dynamic SMT in hard-real-time systems. A closer comparison of “SMT-EDF” and “SMT-ICNT” provides run-time evidence of this safety issue. We find that among 15 unique task-sets scheduled by either “SMT-EDF” or “SMT-ICNT” for the third utilization bin in Figure 5-3(b), 12 task-sets are scheduled by both, 2 are scheduled by only “SMT-EDF”, and 1 is scheduled by only “SMT-ICNT”. The latter 3 task-sets demonstrate that dynamic thread selection affects schedulability, which then raises the deeper concern that any subtle interference can throw things off.

In Figure 5-4, we take a closer look at the third bin of Figure 5-3(a), and show a breakdown of the number of task-sets that are successfully scheduled using “RVMP” and “SMT-EDF”. Of the 25 task-sets, 7 task-sets (28%) are not schedulable by either architecture. Of the 18 task-sets that are successfully schedulable, 14 are schedulable with either by both “RVMP” or and “SMT-EDF”. Interestingly, there are some task-sets that are schedulable using “RVMP” but not “SMT-EDF” (2 task-sets) and vice versa (another 2 task-sets). This observation emphasizes that the behavior of dynamic SMT is totally unanalyzable. Moreover, “RVMP” is capable of safely scheduling some task-sets that fail on the dynamic SMT processor (even with OOO core and dynamic branch prediction).

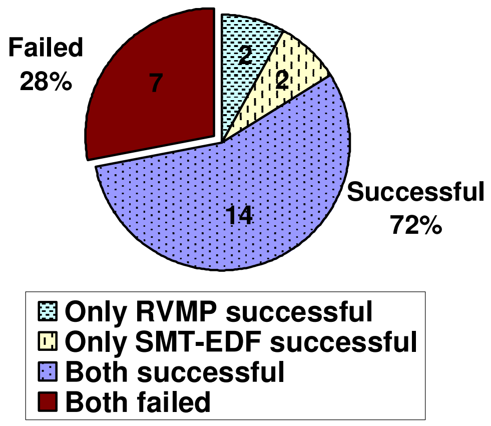


Figure 5-4. Comparison between “RVMP” and “SMT-EDF”.

Chapter 6

Analytical Model for Bounding Memory Overlap on RVMP

In this chapter, the analytical model for statically bounding overlap between memory and computation is presented. Interestingly, this model was developed in the context of a scalar processor *before* RVMP was developed [32]. In hindsight, this earlier work is a special case of RVMP (scalar case), with memory latency tolerance. Here, we extend the latency tolerance aspect to general RVMP (superscalar case), and come full circle to show that the scalar variant is a special case.

The scalar RVMP framework with memory overlap capabilities is indeed an interesting special case. The RVMP schedulability framework in that case is reduced to a simple mathematical closed-form schedulability test (described later), which extends the classic real-time schedulability test of EDF to safely account for memory overlap and improve hard-real-time performance. Our framework provides, for the first time, a safe and tractable infrastructure to use scalar multithreading processors to bridge the processor-memory speed-gap in embedded hard-real-time systems, exceeding the worst-case schedulability limits of classic real-time theory.

The memory overlap analysis starts with the specific case of a single task per virtual processor (Section 6.1), and then is generalized for multiple tasks per virtual processor (Section 6.2).

Note that the analysis is with respect to the number of virtual processors. This provides an abstraction of the underlying hardware that in no way places constraints on either the processor or memory system design. Rather, the opposite is true, *i.e.*, the underlying processor plus memory system implementation dictates the number of available virtual processors and the analytical model is configured accordingly. The number of virtual processors reflects the overall thread-level and memory-level parallelism in the system. Specifically, the number of virtual processors is the minimum of (1) the number of register contexts, (2) the number of pending memory requests (*i.e.*, number of parallel MTUs), and (3) the number of DRAM banks (for parallel DRAM accesses). In Section 6.3.2, we extend the analytical framework to decouple the number of virtual processors from the number of parallel DRAM banks, so that we can capitalize on some overlap opportunity even with limited parallelism in the DRAM. In Section 6.3.1, we describe how to safely account for serialization of transfers on the memory bus.

In Section 6.4, we show that the memory overlap model extends naturally to support systems with non-uniform worst-case memory latency (*e.g.*, systems that support both flash and DRAM).

6.1 Single Task per Virtual Processor

The RVMP framework allows tractably bounding the amount of overlap between memory time of a task and computation time of other tasks by forcibly creating overlap opportunities. This is achieved by the round-based scheduling policy, forcing task switches in a sequence. This concept is illustrated using a scalar RVMP processor, shown in Figure 6-1. Four tasks $T1$, $T2$, $T3$, and $T4$ run on virtual processors $VP1$, $VP2$, $VP3$, and $VP4$, respectively. Each task has possession of the pipeline during its duty cycle each round. By enforcing duty cycles, the WCET of each task is dilated between its release and deadline, as described previously in Section 1.2.2. The duty cycle of each task must be sufficient to complete the dilated task before its deadline on its virtual processor (*i.e.*, satisfy the condition: $dilated\,WCET \leq period$). As such, a safe duty cycle d must be calculated for each task to guarantee that this condition is met.

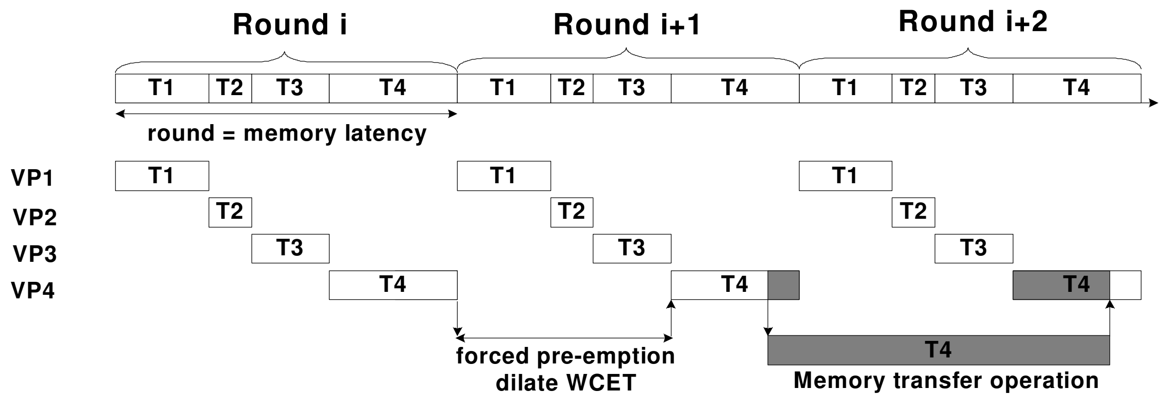


Figure 6-1. Exploiting RVMP for bounding memory overlap.

A forced preemption can occur during computation or during a memory transfer. If it happens during computation, WCET is dilated because the task becomes completely idle, doing neither computation nor a memory transfer. This scenario is highlighted in Figure 6-1 for the first forced preemption of $T4$. However, if a task manages to initiate a memory transfer before being forcibly preempted, the transfer will continue in spite of the forced preemption, thanks to the task's private MTU. The key to this approach is to set the round equal to the latency of a memory transfer. This ensures that a memory transfer, regardless of where it occurs within a task, will begin and end in consecutive duty cycles of the task, as shown for rounds $i + 1$ and $i + 2$ of $T4$ in Figure 6-1. In this way, WCET is *not* dilated by forced preemptions during memory transfers, since finishing a memory transfer is marked by immediate resumption of computation. Moreover, this result holds independent of where memory transfers occur within the task. This is significant because it means we can mathematically model a task as being composed of two separable time components, total computation time C and total memory time M, where C is dilated by forced preemptions but M is not. C is dilated by the inverse of the duty cycle (*e.g.*, if duty cycle = 0.5, computation time doubles). Thus, the WCET expands from $C + M$ to $\frac{C}{d} + M$, where d is the duty cycle.

The key idea of the memory overlap framework is that by taking into account that the memory component is not dilated by the duty cycle, we can eliminate the idle time that arises from over-estimating the duty cycle in the base RVMP framework (Figure 1-6).

In order for the task to safely meet its deadline, the dilated WCET (WCET') must be less than the task's deadline (which is the period in this case):

Equation 6-1. $$WCET'_i \leq period_i \quad \Rightarrow \quad \frac{C_i}{d_i} + M_i \leq period_i \quad \forall \text{ task } i$$

To minimize the utilization of the *physical* processor, each task should be assigned the *minimum* duty cycle d (where $d \leq 1$) that would satisfy Equation 6-1. As such, d can be calculated as follows:

Equation 6-2. $$d_i = \frac{C_i}{period_i - M_i} \quad \forall \text{ task } i$$

Note that, for a scalar RVMP pipeline, there is only one possible superscalar way assignment per task – 1 way each – and tasks purely time-share the pipeline. As such, the bin-packing scheduling framework of Chapter 4 is reduced to a simple closed-form test of whether or not the sum of all tasks' duty cycles is less than or equal to 1.

Equation 6-3. $$\sum_i d_i \leq 1 \quad \Rightarrow \quad \sum_i \frac{C_i}{period_i - M_i} \quad \text{(for scalar RVMP only)}$$

Compare Equation 6-3 to EDF's schedulability condition:

Equation 6-4. $$\sum_i U_i \leq 1 \quad \Rightarrow \quad \sum_i \frac{C_i + M_i}{period_i} \qquad \text{(EDF schedulability test)}$$

Notice the main difference between the two closed-form schedulability tests of Equation 6-3 and Equation 6-4. RVMP has the advantage of statically bounding possible memory overlap and accounting for it in the schedulability test, whereas EDF completely serializes all memory accesses. As such, RVMP can calculate tighter, yet still safe, duty cycles for tasks, improving the overall performance of the system over that of a classic EDF. By analytically accounting for possible memory overlap to improve worst-case performance, the scalar RVMP provides the required infrastructure to safely and tractably use a multithreading processor to bridge the processor-memory speed-gap in hard-real-time systems.

For the general case of superscalar RVMP, the duty cycle calculation of Equation 6-2 is repeated for all possible way assignments per task. The computation component (C) of a task will change with the number of superscalar ways assigned to it (C will increase with fewer superscalar ways), and is obtained by WCET analysis. However, the memory component (M) is independent of way assignment. The duty cycles calculated for each way assignment are then used as inputs to the bin-packing algorithm of Chapter 4, which can be applied with no modification. This is another beauty of the analytical memory overlap technique: it integrates seamlessly with the basic RVMP scheduling framework. The only difference is in the way of calculating the duty cycles, which are used for bin-packing as before.

A more formal and complete derivation of this analytical model, including assumptions for correctness, can be found in Appendix B. Also, in Appendix C, we present a proof that RVMP duty cycles calculated using the analytical memory bounding technique produce a better worst-case utilization than basic RVMP duty cycles that don't distinguish between memory and computation.

6.2 Multiple Tasks per Virtual Processor

When there are multiple tasks on a single virtual processor, their WCETs cannot be overlapped because there is only one register context, one memory transfer unit, etc. That is, a virtual processor is logically a conventional single-threaded uniprocessor. As such, conventional uniprocessor scheduling is required within the virtual processor - we use conventional EDF.

The duty cycle expression in Equation 6-2 is generalized by realizing that a duty cycle d is associated with a virtual processor, not any particular task. WCETs of all tasks on a virtual processor are dilated by that virtual processor's duty cycle d. Only their computation components are dilated, yielding the following condition for EDF schedulability of k tasks on a single virtual processor.

Equation 6-5.

$$\frac{\left(\frac{C_1}{d} + M_1\right)}{period_1} + \frac{\left(\frac{C_2}{d} + M_2\right)}{period_2} + \ldots + \frac{\left(\frac{C_k}{d} + M_k\right)}{period_k} \leq 1$$

Each term in the above expression is the modified (perceived) utilization of a single task, *i.e.*, dilated WCET divided by period, and EDF schedulability is assured if the sum of all tasks' modified utilizations is less than or equal to 1. Using exactly 1 will minimize d. Equation 6-5 can be simplified as follows:

Equation 6-6.

$$\left(\frac{\frac{C_1}{period_1}}{d}+\frac{M_1}{period_1}\right)+\left(\frac{\frac{C_2}{period_2}}{d}+\frac{M_2}{period_2}\right)+\ldots+\left(\frac{\frac{C_k}{period_k}}{d}+\frac{M_k}{period_k}\right)=1$$

Solving for d yields the generalized result below.

Equation 6-7.

$$d=\frac{\sum_{j=1}^{k}\frac{C_j}{period_j}}{1-\sum_{j=1}^{k}\frac{M_j}{period_j}}$$

Note that the specialized Equation 6-2 is consistent with the general form Equation 6-7 for $k = 1$.

For scalar RVMP, the schedulability test of Equation 6-3 still applies, whereas for superscalar RVMP these per-VP duty cycles are again used by the bin-packing algorithm to find a suitable schedule. Different heuristics can be used to group tasks to run on the same VP. Similar to the discussion of Section 4.2, tasks can be grouped based on their ILP. With the addition of memory overlap, and since tasks assigned to the same VP will have their memory components *serialized*, another interesting heuristic is assigning tasks with high memory components to different VPs, and thus, load balance the VPs.

6.3 Modeling the Memory System

6.3.1 Modeling Bus Transfer Time

So far, we have separated WCET into computation (C) and memory (M) components. M only accounts for the raw DRAM access time. However, a bus transfer accompanies every DRAM access, which is not accounted for by either the C or M components. We introduce another WCET component, B, to reflect aggregate bus time of a task: the total time spent by a task transferring its memory blocks to/from DRAM.

Bus transfer requests from multiple virtual processors are serialized on the memory bus. In the worst case, a virtual processor may have to wait for $n - 1$ other transfers to complete before it can own the bus, one for each of the other virtual processors (assuming there are n virtual processors). Thus, in the worst-case, a transfer takes n times as long to complete. The aggregate bus time for the system $n \times B$. Thus, a trade-off is revealed: the aggregate bus time of a task is extended ($n \times B$), but we can overlap aggregate memory time plus bus time of the task with computation of other tasks. The dilated WCET in this case will be:

$$WCET' = \left(\frac{C}{d}\right) + (M + n \times B)$$

and the duty cycle:

Equation 6-8. $$d = \frac{C}{period - (M + n \times B)}$$

Using this model, the round is set equal to the latency of one DRAM access plus the extended ($\times n$) bus transfer time of one memory block.

Notice that such a worst-case bus serialization bound is also required for the base RVMP substrate (which doesn't account for overlapping memory accesses) and for multiprocessor systems with shared memory bus [*e.g.*,86]. If the worst-case bus transfer time is not accounted for, the produced WCETs might be under estimated, which jeopardizes the safety of the system.

6.3.2 Modeling Memory Banks

We first consider the case where the number of virtual processors is equal to the number of DRAM banks. We prevent bank conflicts from occurring by mapping virtual processors to DRAM banks, one-to-one. For example, virtual processor 1 is mapped to bank 1, meaning any tasks that run on virtual processor 1 have their instructions/data allocated to bank 1. Tasks on the same virtual processor are serialized on that virtual processor; hence allocating them to the same bank does not introduce conflicts. Tasks on different virtual processors are prevented from conflicting by ensuring their instructions/data are allocated to different banks, corresponding to the virtual processors. Thus, DRAM parallelism is fully exploited.

Next, we extend our analysis to decouple the number of virtual processors from the number of DRAM banks. Thus, the number of virtual processors is governed only by characteristics of the processor core (namely, number of register contexts and MTUs). If the number of DRAM banks is less than the number of virtual processors, then multiple virtual processors share the same DRAM bank and conflicts may occur. In this case, the memory access latency from the perspective of a virtual processor is extended, in the worst case, by a factor s, where

s is the number of virtual processors sharing a single bank. Each access from the virtual processor assumes that the bank is already busy, and has to wait for $s - 1$ other accesses, in the worst-case, to finish before it can proceed. The total memory component M is thus extended to $s \times M$. We can now express the dilated WCET as:

$$WCET' = \left(\frac{C}{d}\right) + (s \times M + n \times B)$$

and the duty cycle as:

Equation 6-9. $$d = \frac{C}{period - (s \times M + n \times B)}$$

As with bus conflicts, bank conflicts reveal a trade-off: the aggregate memory time is extended to $s \times M$, but we can overlap it with computation of other tasks. Using this model, the round is set equal to the extended ($\times s$) DRAM access latency plus the extended ($\times n$) bus transfer time of one memory block.

Similar to worst-case bus transfer time, serialization due to DRAM bank conflicts must be accounted for in the case of the base RVMP and shared-memory multiprocessors.

6.4 Non-uniform Memory Latency

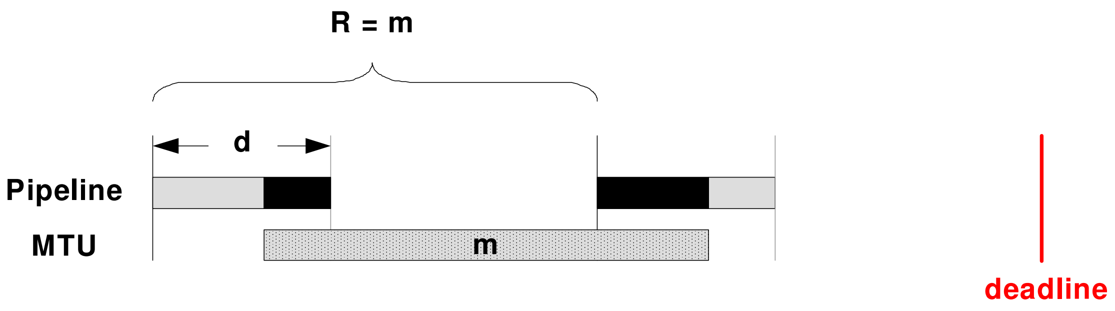
The analysis presented in Section 6.1 naturally extends to support systems with non-uniform memory latency (*e.g.*, where different memory technologies such as DRAM and flash are used in the same system).

Insofar, we set the round length equal to the worst-case memory transfer latency, and derived the mathematical expression of the duty cycle (Equation 6-2) accordingly. We show here that the round length can be set differently, without affecting our mathematical model. We illustrate this with an example.

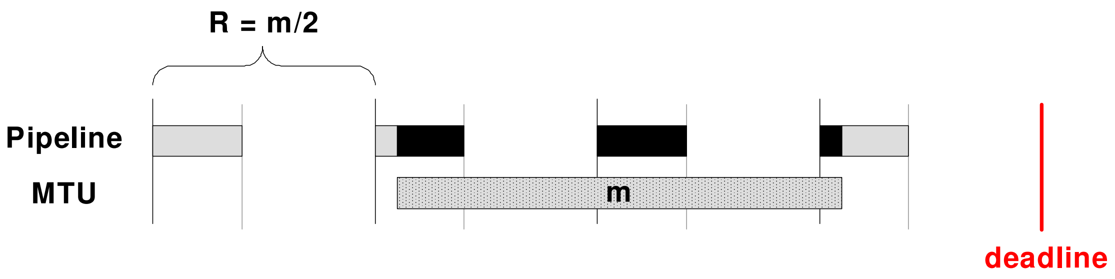
Figure 6-2(a) shows a task with a single memory transfer, m, running using a duty cycle, d, on a scalar RVMP processor. The round length, R, equals the worst-case memory transfer latency, m, which is the standard RVMP approach. During the memory transfer, the pipeline resources are idle during the task's duty cycle, shown in black in Figure 6-2(a). The total idle time, per memory transfer, is the length of a single duty cycle, as described in Section 6.1: Idle time $= d \times R = d \times m$. The task in the example finishes execution within its duty cycle during the second round.

In Figure 6-2(b), the round length is halved ($R = \frac{1}{2} \times m$). This does not change the duty cycle calculated for the task. However, the length of the duty cycle is halved to $d \times \frac{1}{2} \times m$. The amount of idle time is still the same as the previous case (Idle time $= d \times m$), however, it is now distributed over multiple rounds. Similarly, the total pipeline computation of the task is still the same, but it is distributed over multiple rounds. The task still finishes before its deadline in absolute time (which is the end of the last round in the figure). The same thing also applies to Figure 6-2(c), where the round length = $\frac{1}{4}\times$ memory latency.

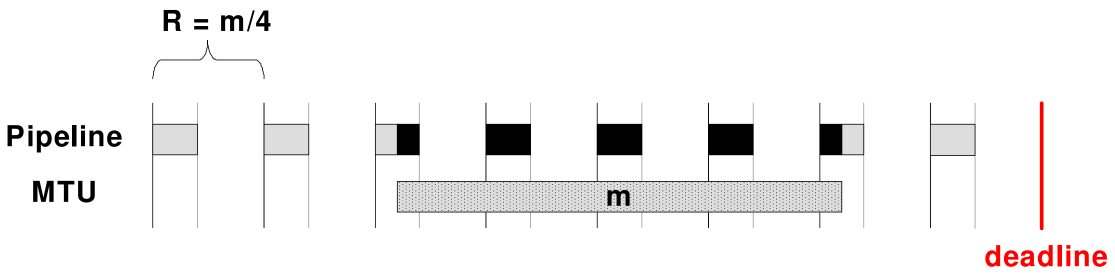
From this example, we can conclude that the round length, R, can be set to be less than the memory latency, m, as long as R is an integer divisor of m (*i.e.*, $\lceil \frac{m}{R} \rceil = \frac{m}{R}$). This in no way affects the correctness of the duty cycle calculation derived earlier, preserving the timing correctness of the approach.



(a) Round = memory latency.



(b) Round = $\frac{1}{2}\times$ memory latency.



(c) Round = $\frac{1}{4}\times$ memory latency.

Figure 6-2. Relation between round length and memory latency.

This observation can be utilized to support multiple memory technologies with different latencies in an RVMP system. The round length can be set as a common integer divisor of the different worst-case latencies. Finding such a common divisor might require the worst-case latencies to be padded, a penalty necessary to achieve better worst-case performance gains by enabling RVMP's safe memory overlap framework.

Chapter 7

Evaluation of Memory Bounding Technique on the RVMP Framework

In this chapter, we evaluate the RVMP substrate with the analytical memory overlap framework derived in Chapter 6. We separate the evaluation into two main parts: first we evaluate superscalar RVMP (Section 7.1) followed by the special case of scalar RVMP (Section 7.2).

7.1 Evaluation of Superscalar RVMP with Memory Overlap

7.1.1 Methodology

The experiments in this section are divided into two main sets. The first set is composed of worst-case schedulability analysis and run-time experiments for task-sets on various architectural models. This approach is the same as that of Chapter 5, except that we now account for possible memory overlap to calculate tighter duty cycles for RVMP according to Equation 6-9, and we model in detail contention for bus bandwidth and conflicts for DRAM banks for all architectural models.

The second set of experiments aims at studying the effect of varying the degree of memory system contention on the schedulability of RVMP (with memory overlap technique ("RVMP (opt)") and without memory overlap technique ("RVMP (non-opt)"), as well as various uniprocessor/multiprocessor configurations with a shared-memory system. To study the effect of memory contention, we vary both the number of parallel DRAM banks (the "s" factor in Equation 6-9) and the worst-case bus serialization (the "n" factor in Equation 6-9).

All the simulation experiments are performed on the same cycle-accurate simulator of Chapter 5, modeling the microarchitecture summarized in Table 5-1.

7.1.1.1 Static Worst-case Timing Analysis

The timing analysis approach used in this chapter is similar to our simulation-assisted manual analysis presented in Section 5.1.1, with one main difference. The optimized duty cycle calculation of Equation 6-9 requires the WCET of a task to be described in terms of 3 components: (1) C: aggregate computation time on the pipeline, (2) M: aggregate memory access time (assuming no conflicts), and (3) B: aggregate bus transfer time (assuming no contention). The computation component (C) is found by simulating the enforced longest program path on our cycle-accurate simulator, as before. To find the memory (M) and bus (B) components, we need the worst-case number of memory transfer instructions in the program, which is readily available either from a separate and orthogonal timing analysis phase or from the number of programmatic memory transfer instructions inserted by the compiler/programmer.

After bounding the computation time component (C) of WCET, we add on the memory time component (M) and the bus time component (B) based on the total number of programmatic memory transfers in the task. We explicitly avoided placing memory transfer instructions in conditional paths (*i.e.*, hammocks), to make aggregate computation time and aggregate memory time of WCET easily separable. If a memory transfer instruction forms one side of a hammock and computation the other side, timing analysis will include the hammock in either C or M depending on which side of the hammock takes more time. However, the two sides are affected differently by duty cycles – M is not dilated whereas C is. Thus, if we allow memory transfer instructions inside hammocks, WCET analysis may have to be modified.

To better understand the problem, we consider a simple example. Figure 7-1(a) shows a simple hypothetical program, composed of four basic blocks: I, II, III, and IV, corresponding to an if-else statement. Block II has an embedded memory transfer instruction, m. The total computation component, C, and total memory component, M, in cycles are listed next to each block. Our analysis framework will detect the path I $\rightarrow$ II $\rightarrow$ IV as the longest path. For simplicity, we assume here that the total worst-case C component is the sum of the individual C components of the basic blocks along that path[1]. As such, along this longest path, $C = 20 + 50 + 10 = 80$, while $M = 100$. If we assume that the period of the task, p, is 300, then the duty cycle for the task is: $d = \frac{C}{p-M} = \frac{80}{300-100} = 0.4$. Assume, however, that at run-time, execution proceeds down the other path: I $\rightarrow$ III $\rightarrow$ IV. The total computation component across that path is: $C = 20 + 100 + 10 = 130$, and the dilated WCET of the task is: $\frac{C}{d} = \frac{130}{0.4} = 325$, which is greater than the task's period, meaning that the task will miss its deadline.

[1]Note that this is not accurate in general: our simulation methodology tightly bounds possible pipeline computation overlap across basic blocks.

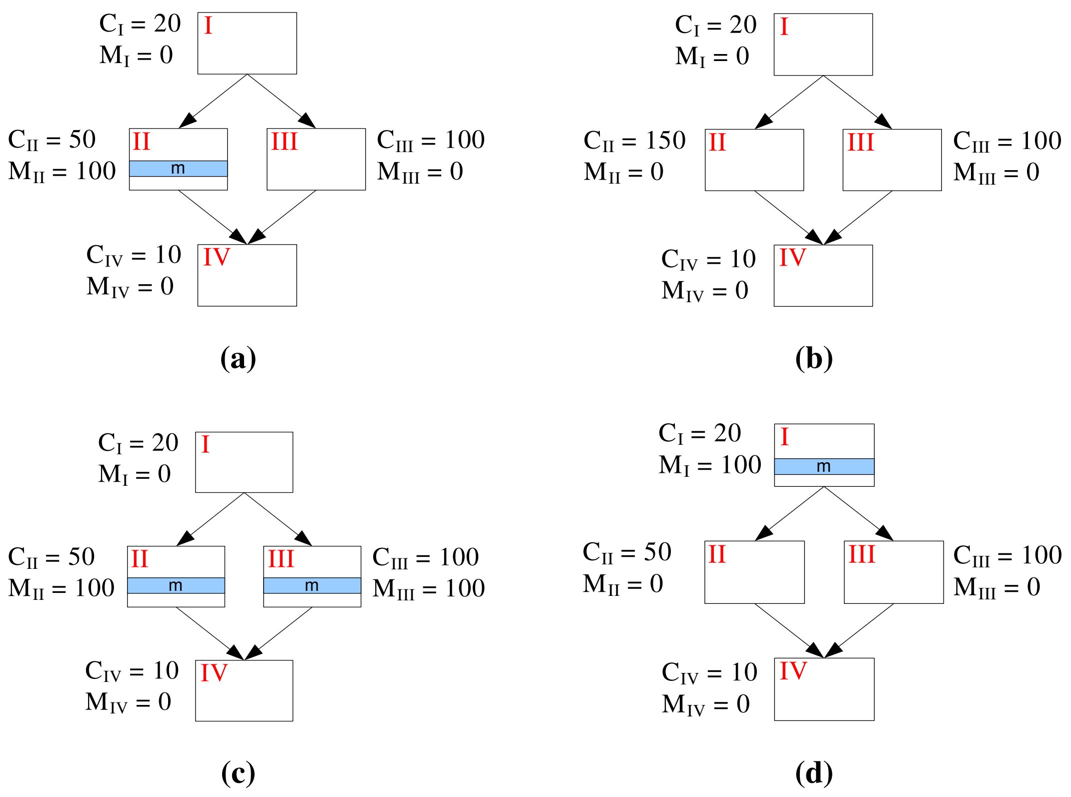


Figure 7-1. Memory transfer instructions in conditional blocks.

A simple solution is to always convert memory transfers embedded in conditional branches to C component, as shown in Figure 7-1(b), trading overlap opportunity for safe analysis. In this case, the worst-case path is still I $\rightarrow$ II $\rightarrow$ IV, but the duty cycle now is: $d = \frac{20+150+10}{300} = \frac{180}{300} = 0.6$, which is safe on either path.

Another simple solution is to logically include the memory transfer in both conditional branches (Figure 7-1(c)), which has the safe effect of moving the transfer latency in series with and out of the hammock (Figure 7-1(d)). In this case, the worst-case path is I $\rightarrow$ III $\rightarrow$ IV, and the duty cycle now is: $d = \frac{20+100+10}{300-100} = \frac{130}{200} = 0.65$, which is still safe on either path.

Finally, worst-case timing analysis could be explicitly modified to work in tandem with the RVMP scheduling, taking into account the duty cycle of a task (WCET parameterized in terms of duty cycle). In our experience with explicit management of scratchpads, we found it unnecessary and over-complicated to embed transfers in hammocks.

7.1.1.2 Modeling Worst-Case Memory Conflict and Bus Contention

We mentioned earlier in Section 6.3 that modeling the worst-case conflicts for DRAM banks and worst-case contention for bus bandwidth is also required for classic multiprocessor systems with shared memory. However, in some cases the worst-case bound will differ for an RVMP processor from that of a multiprocessor system, simply because the number of active VPs/processors that can initiate parallel memory transfers is different. This is illustrated by an example below.

Figure 7-2(a, b, and c) compares the worst-case bus contention for three equivalent models: a single RVMP processor with 4 VPs, a multiprocessor with four 1-way processors, and a multiprocessor with two 2-way processors, respectively. All models have a main memory consisting of four parallel DRAM banks. As such, there is no conflicts for DRAM banks, and the sharing factor, s, of Equation 6-9 for all models is 1, as indicated on the figure. All the models have a shared memory bus. The worst-case number of bus sharers for the RVMP processor (Figure 7-2(a)) is four ($n = 4$), because any of the four VPs can initiate a memory transfer independently. Similarly, for the multiprocessor system with four processors (Figure 7-2(b)), $n = 4$. However, for the multiprocessor model with two processors (Figure 7-2(c)), there are only two possible parallel memory transfer initiators, and as such, $n = 2$. Clearly, this model has an advantage over both the RVMP processor and the four processors in terms of contention for bus bandwidth, which is correctly modeled in the experiments that follow in this chapter.

In Figure 7-3, we show the same three models, however, the memory system now has only a single DRAM bank. Multiple VPs/processors share this single bank, and we must account for the possible serialization of memory accesses at this bottleneck. For both the RVMP (Figure 7-3(a)) and the multiprocessor with four processors (Figure 7-3(b)) the worst-case number of sharers is 4, and thus, the DRAM sharing factor, s, is also 4, as shown in the figure. However, for the multiprocessor model with two processors (Figure 7-3(c)), there are only two sharers of the single DRAM bank, which means that $s = 2$. Similar to bus contention, this advantage in DRAM conflicts is correctly modeled for the remaining experiments.

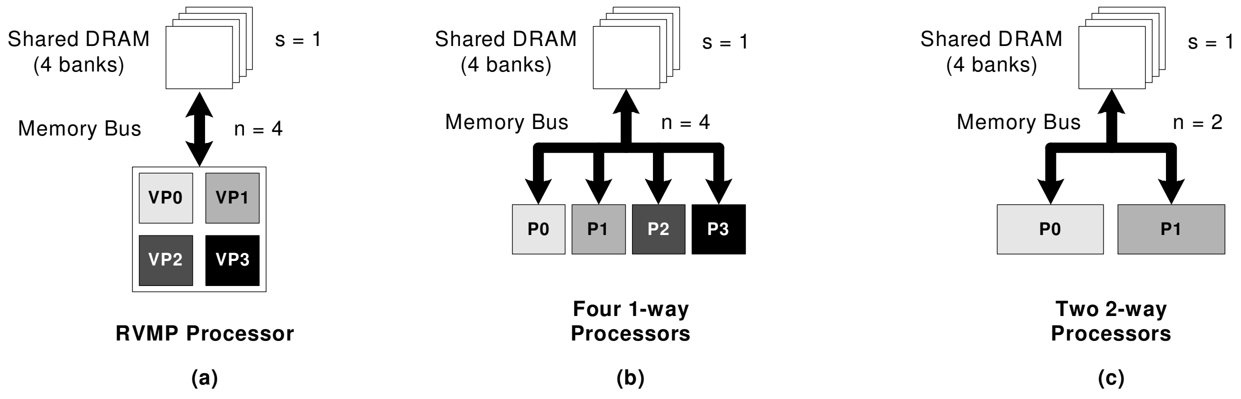


Figure 7-2. Worst-case bus contention for RVMP and multiprocessor models.

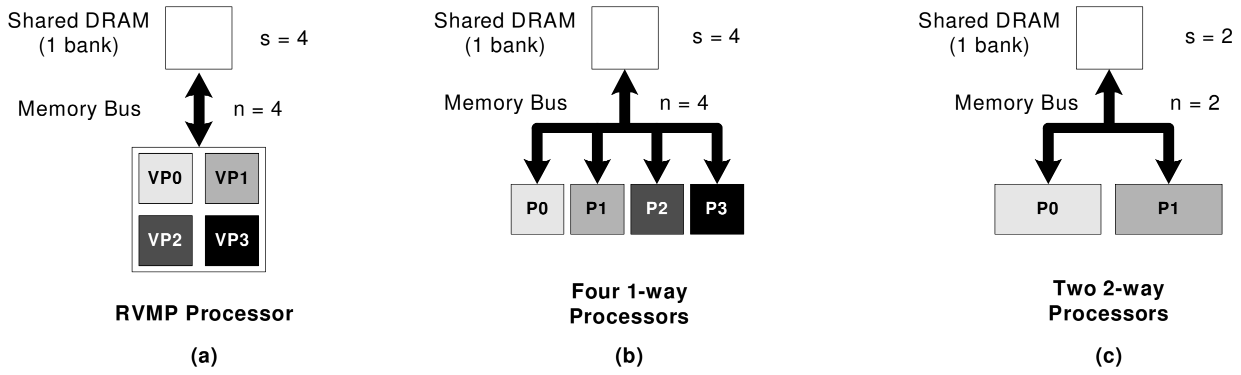


Figure 7-3. Worst-case DRAM contention for RVMP and multiprocessor models.

7.1.1.3 Real-Time Tasks and Task-sets

We use the same C-lab and MiBench benchmarks of Chapter 5. The benchmarks are shown in Table 7-1, and their WCETs are broken into three components (C, M, and B). For each benchmark, the second, third, and fourth columns of Table 7-1 show the computation component of the benchmark assuming it is running on a 1-way, 2-way, and 4-way in-order non-RVMP processor, respectively (as a part of multiprocessor system). The next four columns specify the computation component of the task on a 1-way, 2-way, 3-way, and 4-way RVMP processor, respectively. Notice that C_1 and C_2 on a non-RVMP processor are less than the corresponding C_1 and C_2 on an RVMP processor. This is due to the fact that function units in an RVMP processor are shared according to a worst-case wait fashion to guarantee static analyzability and predictability, extending the C component of the task, as described in Section 3.4. The non-RVMP processor models, as well as the 4-way RVMP model, do not need to extend the execution time, because there can be only one active thread at the same time (no conflict for function units).

The next two columns of Table 7-1 show the memory component of the task broken into M (no conflict aggregate memory access time) and B (no contention bus access time). The last column shows the total number of programmatic memory transfer instructions in each benchmark.

The memory component (M) of a task i is calculated according to the following expression:

$$M_i = (\text{\# of mem. transfer instr.})_i \times \text{DRAM access time}$$

For example, since the worst-case DRAM access time is 50 ns (given in Table 5-1), for the benchmark CNT, the aggregate memory component $M = 441 \times 50$ ns $= 0.0220$ ms.

Similarly, the bus component (B) of a task i is calculated as follows:

$$B_i = (\text{\# of mem. transfer instr.})_i \times \frac{\text{block size}}{\text{bus width}} \times \frac{1}{f_{bus}}$$

The size of a single memory block is 128 bytes, the bus width is 4 bytes, and the bus frequency is 500 MHz (again, all given in Table 5-1). The bus component of task CNT is: $441 \times \frac{128}{4} \times \frac{1}{500} = 0.0282$ ms.

Task-sets with eight tasks each are generated in a similar way to Section 5.1.2. Random task periods are selected according to non-RVMP WCETs, such that: $WCET_8 \leq period \leq 8 \times WCET_1$. Recall that, the scalar utilization (U_{scalar}) of a task-set is calculated as: $\sum_\tau \frac{WCET_{\tau,1}}{period_\tau}$ (calculated using the non-extended C_1 component of the task). Task-sets are grouped into four bins according to their U_{scalar}. The four bins are as follows: $0 < U_{scalar} \leq 1$, $1 < U_{scalar} \leq 2$, $2 < U_{scalar} \leq 3$, and $3 < U_{scalar} \leq 4$. Each bin has 25 randomly-generated task-sets. These bins represent increasing difficulty in scheduling a task-set, the first bin containing the least demanding task-sets and the fourth bin containing the most demanding task-sets.

Table 7-1. Benchmarks
WCET broken down to C, M, and B ($f_{proc} = 1$GHz, $f_{bus} = 500$MHz).

Task	**Non-RVMP C (ms)**			**RVMP C (ms)**				**Memory (ms)**		**# of mem. transfer instr.**
	C_1	C_2	C_4	C_1	C_2	C_3	C_4	M	B	
CNT	0.0508	0.0373	0.0274	0.0677	0.0426	0.0274	0.0274	0.0220	0.0282	441
ADPCM	2.51	2.05	1.58	3.00	2.23	1.80	1.58	0.0256	0.0328	512
SRT	1.95	1.31	1.01	2.55	1.51	1.13	1.01	0.00200	0.00256	40
MM	1.31	1.67	1.18	2.14	1.50	1.18	1.18	0.345	0.442	6908
FFT	0.390	0.290	0.233	0.478	0.312	0.282	0.233	0.0936	0.120	1873
LMS	0.152	0.101	0.0904	0.198	0.134	0.117	0.0904	0.00265	0.00339	53
CRC	0.0240	0.0203	0.0160	0.0320	0.0239	0.0160	0.0160	0.0120	0.0154	240
TOAST	0.194	0.146	0.122	0.237	0.152	0.144	0.122	0.0481	0.0616	962
LAME	7.72	6.02	5.10	8.62	6.47	5.78	5.10	0.516	0.654	10230

7.1.2 Results

7.1.2.1 Worst-Case Schedulability Analysis

The graph in Figure 7-4 shows the worst-case schedulability results for various statically analyzable architectural models. Each task-set is composed of 8 tasks, with two tasks sharing each VP for RVMP. The task-sets are the same task-sets used in Figure 5-1(b). For each utilization

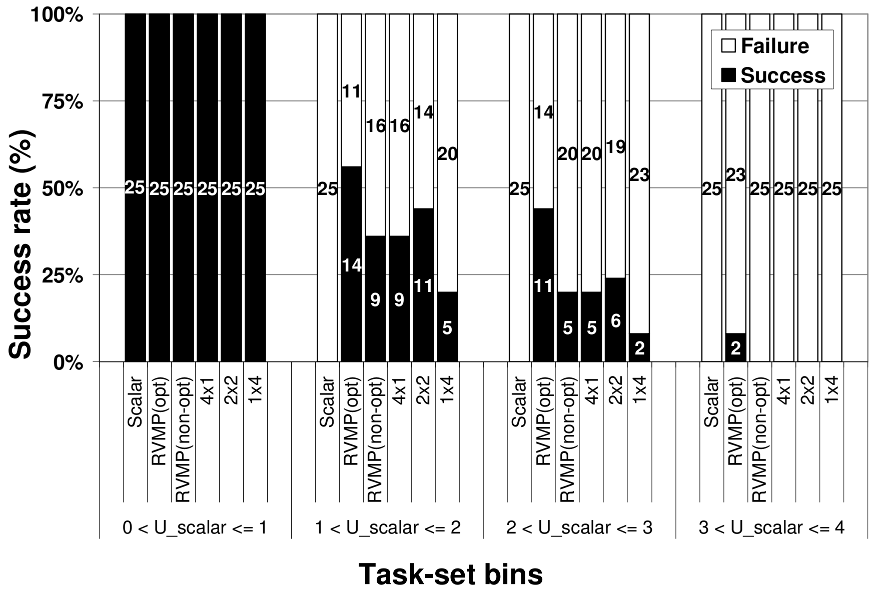


Figure 7-4. Worst-case schedulability analysis (8 tasks per task-set).

bin, we plot (on the y-axis) the number of tasks that were successfully schedulable ("Success") versus not schedulable ("Failure") for the various architectures. "Scalar" is the base in-order scalar processor used to calculate U_{scalar}. "RVMP (opt)" is our proposed RVMP substrate that uses the *optimized* duty cycle calculation of Equation 6-9. "RVMP (non-opt)" is the base RVMP framework that doesn't distinguish between serial pipeline computation and parallelizable memory accesses. The other three bars correspond to classic earliest-deadline-first

(EDF) scheduling on various conventional uniprocessor and multiprocessor configurations: "4$\times$1" (four in-order scalar processors), "2$\times$2" (two in-order 2-way superscalar processors), and "1$\times$4" (a single in-order 4-way superscalar processor). Notice that, "RVMP (non-opt)" of Figure 7-4 is the same as "RVMP" of Figure 5-1, but with one main difference: "RVMP (non-opt)" models the worst-case contention for memory bandwidth. Similarly, "4$\times$1" and "2$\times$2" have to account for worst-case bus access time assuming 4 and 2 processors are sharing the bus, respectively, as described in Section 7.1.1.2. This explains why the schedulability success rates of these models in Figure 7-4 is less than the success rates of Figure 5-1. "1$\times$4" doesn't suffer from this penalty, because there is only one processor in the system, owning all the bus bandwidth ($n = 1$).

For the experiments in this section, we assume that there are four parallel DRAM banks (*i.e.*, $s = 1$ in Equation 6-9). The worst-case bus sharing extension (n in Equation 6-9) is assumed to be the worst-case number of active processors/virtual processors that can initiate memory transfers in parallel in the overall system, as described in Section 7.1.1.2. For example, for the "2$\times$2" model (two processors), the worst-case number of bus sharers, n, is 2 (each of the two processors can initiate memory transfers in parallel). Similarly, for an RVMP processor, the worst-case n is 4 (a maximum of 4 active VPs).

The general observations and conclusions we made in Section 5.2.1 about the worst-case schedulability of task-sets on the various architectural models are still valid here. Of specific importance, however, is comparing "RVMP (opt)" to the other models. Notice that, "RVMP (opt)" significantly outperforms all other models in static worst-case schedulability. This is especially interesting, taken into account that the multiprocessor models have the advantage of unextended C component of WCET (shown in Table 7-1), in addition to a lower worst-case bus contention factor (n) for the "2×2" ($n = 2$) and "1×4" ($n = 1$) models. For the highest three utilization bins in Figure 7-4, "RVMP (opt)" successfully scheduled 27 task-sets out of a total of 75 task-sets (36% success rate), compared to 14 task-sets for "RVMP (non-opt)" (19% success rate), 14 task-sets for "4×1" (19% success rate), 17 task-sets for "2×2" (22.6 %), and 7 task-sets for "1×4" (9.3%). The main advantage of "RVMP (opt)" is its ability to exploit statically bounded overlap between pipeline computation and memory accesses from different tasks to compute tighter duty cycles, improving the overall worst-case schedulability of the system. Moreover, both RVMP models are capable of flexibly choosing multiple processor configurations in a single round by assigning the best number of superscalar ways to each task in order for the task-set as a whole to be schedulable. This flexibility is not available to the uniprocessor/multiprocessor systems, explaining the better worst-case schedulability performance of "RVMP (opt)" over the multiprocessor systems.

An interesting observation is that "2×2" outperforms "RVMP (non-opt)" in most cases. The worst-case bus extension factor, n, for "RVMP (non-opt)" (as well as "RVMP (opt)") is 4, compared to a factor of 2 for "2×2", as shown in Figure 7-2. This significantly increases the M component of the WCET (in addition to the extended C component described above). Unlike "RVMP (opt)", "RVMP (non-opt)" does not have the advantage and capability to exploit memory overlap to improve worst-case schedulability. As a result, "RVMP (non-opt)" relies solely on its flexibility in partitioning and reconfiguring the processor to achieve better worst-case schedulability than the rigid multiprocessors. In some cases, however, no valid configuration can be found, whereas a multiprocessor system can schedule the task-set because its execution time is not extended, outperforming “RVMP (non-opt)” in such cases.

Notice that, “1×4” has an advantage over all other models in terms of worst-case bus serialization. Since “1×4” is single threaded, there can be only one bus sharer at any time, which means that the bus component, B, of WCET is not extended. However, since “1×4” is scalar, its schedulability success rate is considerably lower than that of all other models, because a 4-way in-order single-threaded scalar processor cannot capitalize on neither ILP nor TLP.

7.1.2.2 Run-time Experiments

In Figure 7-5, we show the number of task-sets that succeed or fail at run-time on the various architectures, using our cycle-level simulator. A task-set is considered successful if all deadlines are met for the simulated time-frame (the lesser of the hyper-period or 100 ms). To reiterate, whereas *formal* schedulability results from the previous subsection hold in the *worst case*, regardless of the task-set inputs, simulation results in this section only hold for *specific* task-set inputs.

We compare the statically analyzable models with the same unsafe dynamic SMT models: “SMT-EDF” and “SMT-ICNT”. Recall from Section 5.2.2 that “SMT-EDF” uses a real-time-aware (but still not provably safe) thread selection policy that prioritizes tasks according to earliest deadlines, while “SMT-ICNT” uses the ICOUNT thread selection policy.

The run-time performance of the statically analyzable architectural models is in agreement with the formal schedulability test performed earlier. The slight difference between

Figure 7-5 and Figure 7-4 is due to the difference between worst-case execution time estimates and actual execution time.

When contention for memory bus bandwidth is properly modeled, only "RVMP (opt)" can approach the performance of the dynamic (and unsafe) SMT models. For the highest three utilization bins, "SMT-EDF" and "SMT-ICNT" were dynamically successful in scheduling 34 and 31 task-sets out of 75, respectively (45% and 41% success rate, respectively), compared to 27 task-sets for "RVMP (opt)" (36% success rate). The performance gap between the dynamic SMT models and "RVMP (non-opt)" has significantly increased after modeling contention for memory bus bandwidth. For the highest three utilization bins, the difference in the success rate of "SMT-EDF" and "RVMP (non-opt)" increased from 1% with no bus contention (Figure 5-3(b)) to around 25% with bus contention (Figure 7-5). The same thing applies also for the other multiprocessor configurations.

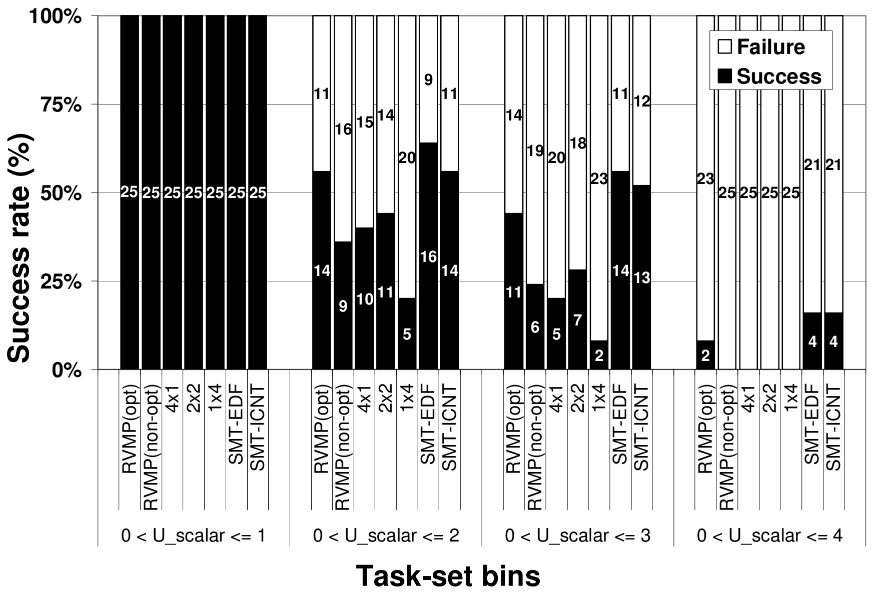


Figure 7-5. Run-time experiments (8 tasks per task-set).

7.1.2.3 Understanding Memory Overlap on RVMP

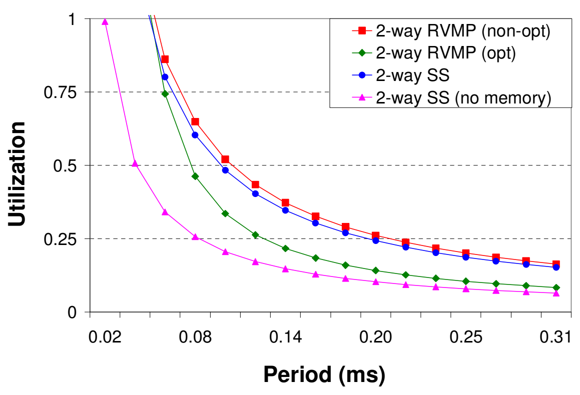
To better understand the behavior of memory overlap on RVMP, we classify the benchmarks of Table 7-1 according to their memory-to-computation ratio ($\frac{M+B}{C}$). Table 7-2 lists the memory-to-computation ratio for each task according to its 1-way, 2-way, 3-way, and 4-way computation component, respectively. Tasks are grouped into two categories according to their 1-way memory-to-computation ratio: (1) High: tasks with memory-to-computation ratio greater than 0.25 (CRC, CNT, FFT, MM, and TOAST), and (2) Low: tasks with memory-to-computation ratio less than 0.25 (LAME, LMS, ADPCM, and SRT).

Table 7-2. Benchmark memory-to-computation ratio ($n = 1, s = 1$).

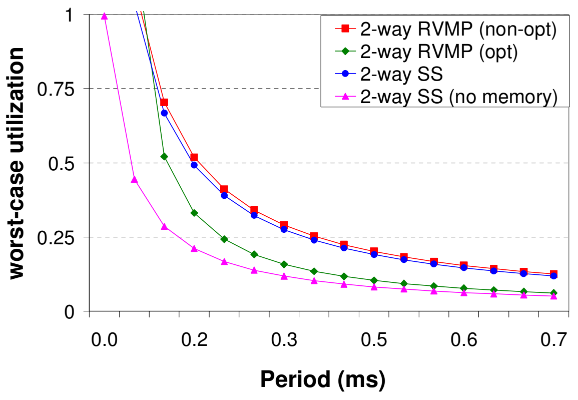
	Task	1-way	2-way	3-way	4-way
High	CRC	0.855	1.14	1.71	1.71
	CNT	0.742	1.18	1.83	1.83
	FFT	0.447	0.684	0.757	0.916
	MM	0.367	0.523	0.665	0.665
	TOAST	0.334	0.721	0.761	0.899
Low	LAME	0.135	0.180	0.201	0.228
	LMS	0.0305	0.0451	0.0516	0.0668
	ADPCM	0.0194	0.0261	0.0324	0.0369
	SRT	0.00178	0.00302	0.00405	0.00453

Tasks within each of these two groups have similar worst-case utilization profile. We look at these tasks in detail in Figure 7-6 (tasks in the High category) and Figure 7-7 (tasks in the Low category).

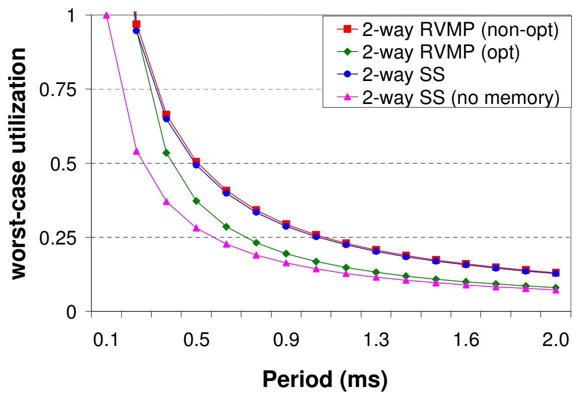
In Figure 7-6, we plot the worst-case utilization of each task (y-axis) vs. its period (x-axis), assuming no DRAM conflicts ($s = 1$) and no bus contention ($n = 1$). The utilization of a task is independent of its workload context. The utilization is shown for 4 different architectural



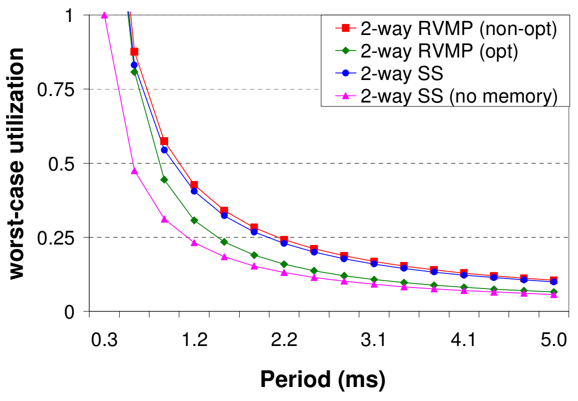
(a) CRC



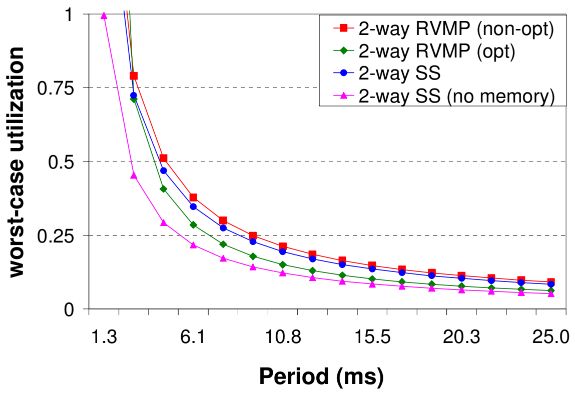
(b) CNT



(c) TOAST



(d) FFT



(e) MM

Figure 7-6. Worst-case utilization of benchmarks with high memory-to-computation ratio on different models. No bus contention ($n = 1$), no DRAM conflicts ($s = 1$).

models. “2-way RVMP (non-opt)” is a 2-way RVMP VP that uses the non-optimized duty cycle (no memory overlap). “2-way RVMP (opt)” is a 2-way RVMP VP that uses the optimized duty cycle (with memory overlap). “2-way SS” is a 2-way superscalar processor (part of a “2×2” multiprocessor system). “2-way SS (no memory)” represents a lower utilization bound in an ideal 2-way superscalar processor, where all the memory component of a task is hidden (*i.e.*, $M = 0, B = 0$). Notice that, for the two RVMP models, the worst-case utilization is actually the task’s duty cycle. The lower limit of the period (on the x-axis) is set such that the worst-case utilization of the ideal “2-way SS (no memory)” is 1, while the upper limit is set such that the utilization of “2-way SS (no memory)” is 0.05, as evident from the plots. This range of periods covers the relevant utilization range of the task. The lower limit of the period represents the ideal highest schedulable utilization. Any higher utilization will be unschedulable for all models, and thus is irrelevant for this study. Similarly, the higher limit of the period represents a reasonably low utilization task. Further reducing the utilization of the task will not add any significant insight to the experiments.

We show the utilization of the tasks using only 2-way models to make the graphs more readable. The utilization profile for other number of ways follows the same trends as that of the 2-way presented here.

We can make three key observations regarding the results in Figure 7-6, valid for all tasks:

1. “RVMP (opt)” is the only model capable of approaching the ideal performance of “SS (no memory)”. However, the utilization of “RVMP (opt)” cannot reach the ideal utilization exactly, because there is still an exposed component of memory that cannot be

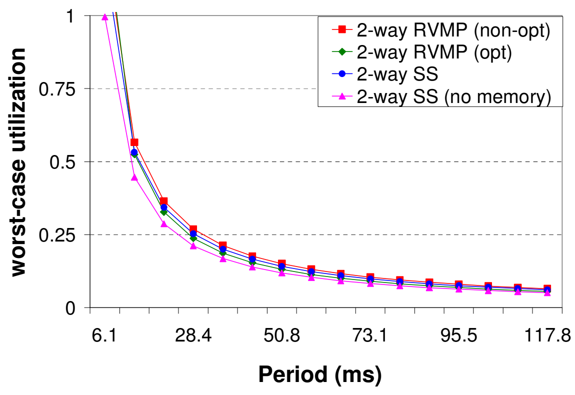
overlapped. This is the amount of idle time during the task's duty cycle when a memory transfer is going on, shown in Figure 6-1.

2. "RVMP (opt)" consistently outperforms "SS" for the same memory system parameters. This is especially interesting, knowing that "RVMP (opt)" extends the worst-case computation component of the task to safely share function units among simultaneous tasks. The ability of "RVMP (opt)" to calculate tighter duty cycles by accounting for memory overlap more than compensates for the inflated execution time. Moreover, the performance gap between "RVMP (opt)" and "SS" increases as the memory-to-computation ratio of the task increases.

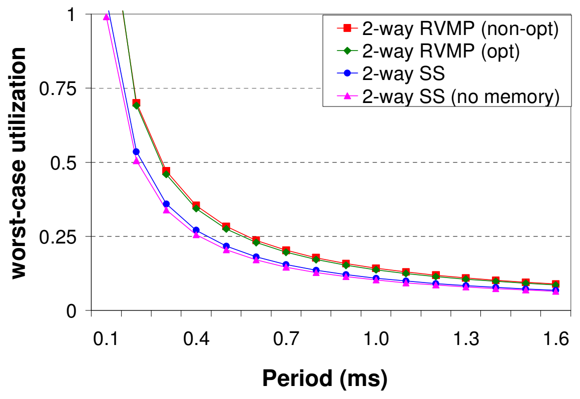
3. The performance of "SS" is consistently better than that of "RVMP (non-opt)". Unlike "RVMP (opt)", "RVMP (non-opt)" does not account for memory overlap, and as such, does not have a performance advantage to mitigate the effect of extended computation components. However, for these benchmarks with high memory-to-computation ratio, the performance difference between "RVMP (non-opt)" and "SS" is very minimal. This is because the memory component of the task dominates its worst-case execution time, making the extended computation component effect on utilization almost negligible. For example, for the CRC benchmark, the difference between the extended and unextended computation component for 2-ways is: $0.0239 - 0.0203 = 0.0036$ ms (from Table 7-1). This is almost an 18% increase in the computation component. However, this increase is not directly translated into an 18% increase in worst-case utilization. The memory component of the task is $s \times M + n \times B = 1 \times 0.0120 + 1 \times 0.0154 = 0.0274$ ms, and

the total unextended WCET is $0.0274 + 0.0239 = 0.0513$ ms. The extended computation penalty of 0.0036 ms is only around 7% of the total WCET. This explains the small difference in utilization between the two models.

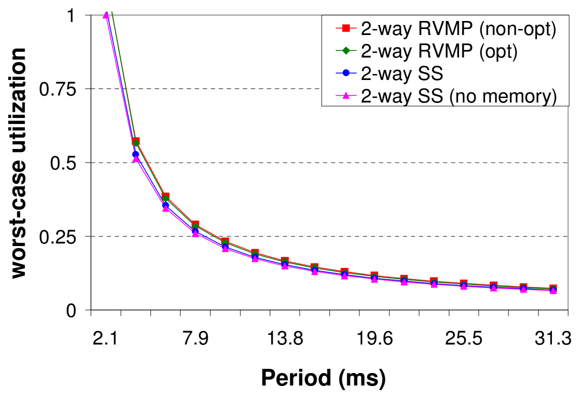
For benchmarks with low memory-to-computation ratios (Figure 7-7), the situation is significantly different. With no memory component to overlap, the performance advantage of “RVMP (opt)” diminishes, where its performance almost reaches that of “RVMP (non-opt)”. Similarly, the performance of “SS” is almost that of “SS (no memory)”. Moreover, since the memory component of WCET is very small, the performance difference between the RVMP models and “SS” increases, because the extended computation penalty becomes now more dominant. However, the performance of RVMP should not be assessed based on the utilization of a single task. On the contrary, the power of the RVMP framework is revealed in the context of a whole task-set, where the flexibility of RVMP’s partitioning and reconfiguration pays off, as we demonstrated in Section 7.1.2.



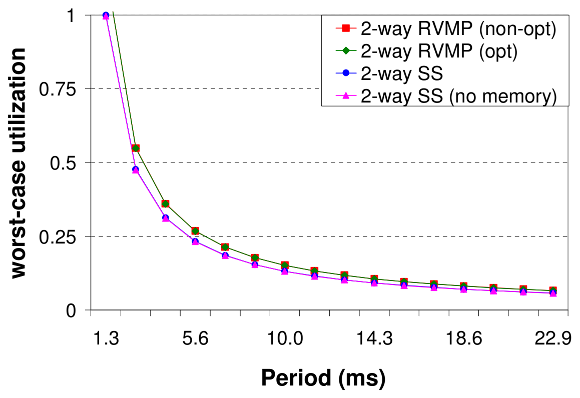
(a) LAME



(b) LMS



(c) ADPCM



(d) SRT

Figure 7-7. Worst-case utilization of benchmarks with low memory-to-computation ratio on different models. No bus contention ($n = 1$), no DRAM conflicts ($s = 1$).

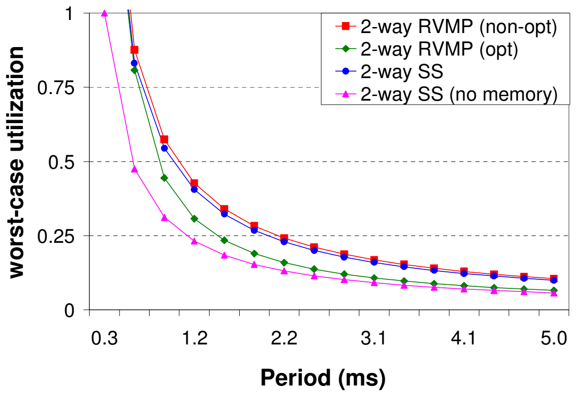
7.1.2.4 Effect of Bus Contention on Schedulability

In this section, we study the effect of varying the worst-case bus contention on the schedulability of RVMP. For all the results in this section, we assume that there is no conflict for DRAM banks ($s = 1$). Since the schedulability of benchmarks with low memory-to-computation ratios is negligibly affected by varying the memory system parameters, we limit our discussion here to benchmarks with large memory-to-computation ratios. Moreover, since the behavior of these benchmarks follow the same trends, we will show results for only one representative benchmark, which is FFT.

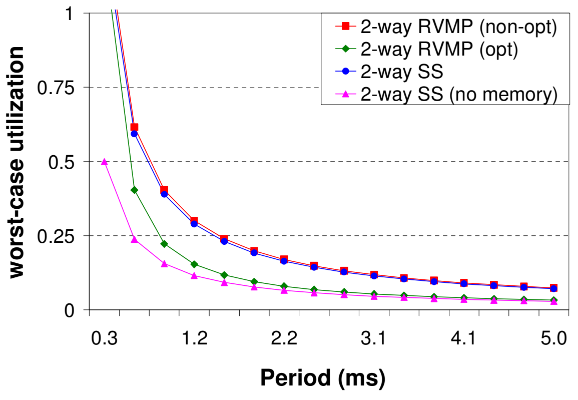
Figure 7-8 shows a comparison of the worst-case utilization of FFT for different periods on the same four architectural models described in Section 7.1.2.3, with 2-ways each. The lower limit of the period (on the x-axis) is set such that the worst-case utilization of “2-way SS (no memory)” is 1, while the upper limit is set such that the utilization of “2-way SS (no memory)” is 0.05.

The worst-case bus contention factor, n, is set to 1, 2, and 4 in Figure 7-8 (first, second, and third row of the figure, respectively). Increasing the value of n increases the contribution of the task’s memory component to its WCET.

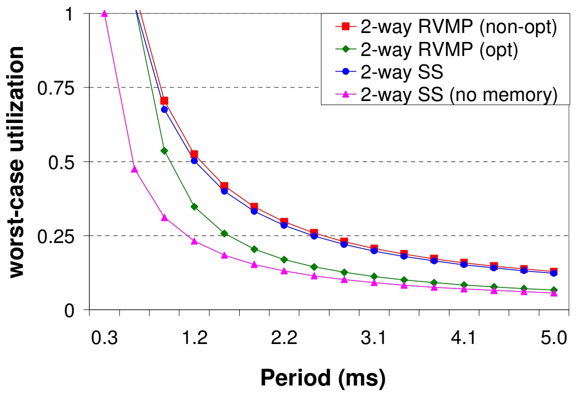
For $n = 1$ (first row of Figure 7-8), the ratio of the processor frequency to the bus frequency ($\frac{f_{proc}}{f_{bus}}$) is changed from 2 (Figure 7-8(a)) to 4 (Figure 7-8(b)) by doubling the processor frequency. Doubling the processor frequency does not affect the memory component of a task, but it reduces the computation component of a task by half. That’s why the utilization of “SS (no memory)” was halved from 1 (Figure 7-8(a)) to 0.5 (Figure 7-8(b)) for the smallest period (left-most point on the graph). This is not true for the other models, because the memory component is not scaled down as well. In fact, doubling $\frac{f_{proc}}{f_{bus}}$ exaggerates the processor-memory performance gap, making overlapping memory even more valuable. This can be seen from the figure: for example, the difference in utilization between “RVMP (opt)” and “SS”, at a period of 1.2 ms, increases from 10% for $\frac{f_{proc}}{f_{bus}} = 2$ (Figure 7-8(a)) to around 15% for $\frac{f_{proc}}{f_{bus}} = 4$ (Figure 7-8(b)). Similarly, the performance gap between “RVMP (opt)” and the ideal “SS (no memory)”, at a period of 1.2 ms, decreases from around 8% (Figure 7-8(a))



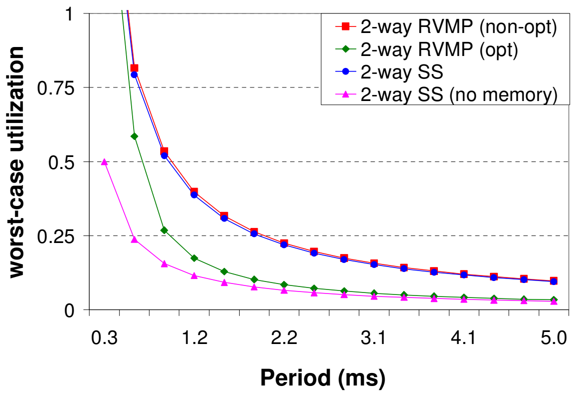
(a) $n = 1, \frac{f_{proc}}{f_{bus}} = 2$



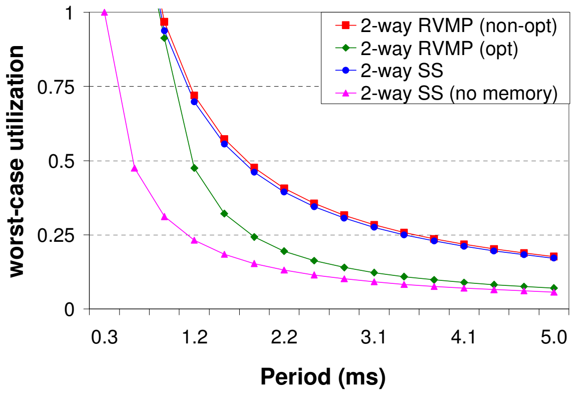
(b) $n = 1, \frac{f_{proc}}{f_{bus}} = 4$



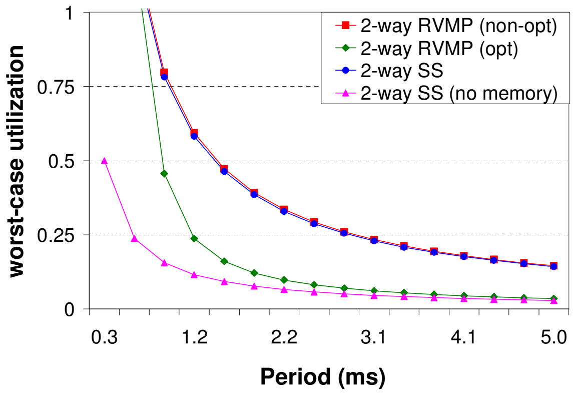
(c) $n = 2, \frac{f_{proc}}{f_{bus}} = 2$



(d) $n = 2, \frac{f_{proc}}{f_{bus}} = 4$



(e) $n = 4, \frac{f_{proc}}{f_{bus}} = 2$



(f) $n = 4, \frac{f_{proc}}{f_{bus}} = 4$

Figure 7-8. Worst-case utilization of FFT: varying bus contention ($n = 1, 2, 4$), no DRAM conflict ($s = 1$).

to 4% (Figure 7-8(b)). This illustrates that the advantage of "RVMP (opt)" increases as the performance difference between processor and memory increases.

Another thing to note is that since the computation component of the task is halved by doubling the processor frequency, the total WCET of the task becomes smaller. As such, the task becomes easier to schedule. This is also evident from the graph: all curves are slightly shifted to the left in Figure 7-8(b) compared to Figure 7-8(a). This means that more tasks with smaller periods (*i.e.*, tighter deadlines) become schedulable (*i.e.*, $U \leq 1$).

In the second row of Figure 7-8, the worst-case bus contention factor is increased to 2, increasing the memory-to-computation ratio of the task even more. For the same $\frac{f_{proc}}{f_{bus}}$ (*e.g.*, Figure 7-8(b) and Figure 7-8(d)) the performance gap between "RVMP (opt)" and "SS" increases, to the advantage of "RVMP (opt)". Note that, however, increasing n significantly increases the WCET of a task, making tasks more difficult to schedule (curves are shifted to right compared to the first row of the figure). There is one exception: since "SS (no memory)" model does not have any memory component, its schedulability is not affected by changing the memory system contention. The "SS (no memory)" curve stays constant for all values of n at the same $\frac{f_{proc}}{f_{bus}}$ ratio. This also means that the performance gap between "RVMP (opt)" and "SS (no memory)" is increased. Similar observations can be made regarding the third row of Figure 7-8 ($n = 4$).

In Figure 7-9, we plot the worst-case utilization of FFT for the worst-case bus serialization scenario for RVMP and superscalar multiprocessors, in the absence of DRAM conflicts ($s = 1$). Recall from Section 7.1.1.2 that for an RVMP processor, the worst-case number of bus sharers is 4 (*i.e.*, $n = 4$), while for a multiprocessor with two processors, the worst-case number of bus sharers is 2 ($n = 2$). This gives an advantage for the multiprocessor over RVMP in terms of extended worst-case memory component due to bus serialization ($n \times B$). This is evident from Figure 7-9: for a small range of periods ($period \leq 1.2$ ms), the worst-case utilization of "SS" is *less* (*i.e.*, better) than that of "RVMP (opt)". Even with its ability to hide memory latency, the doubled worst-case bus component of "RVMP (opt)" overshadows the performance benefits over "SS" for that high utilization range. However, for a considerable range of period ($period > 1.2$ ms), where the utilization of the task is less than 50% of the overall system utilization, "RVMP (opt)" considerably outperforms "SS" even with the increased bus penalty. Note that, since "RVMP (non-opt)" does not overlap memory, the performance gap between it and "SS" increases significantly.

Notice that RVMP has this disadvantage only against a multiprocessor system with less than four processors. The worst-case bus serialization factor, n, for a multiprocessor with four processors is 4, which is the same as that of RVMP.

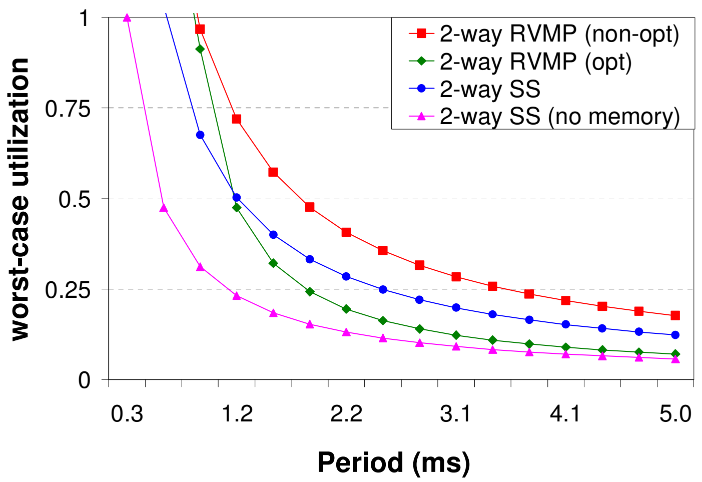
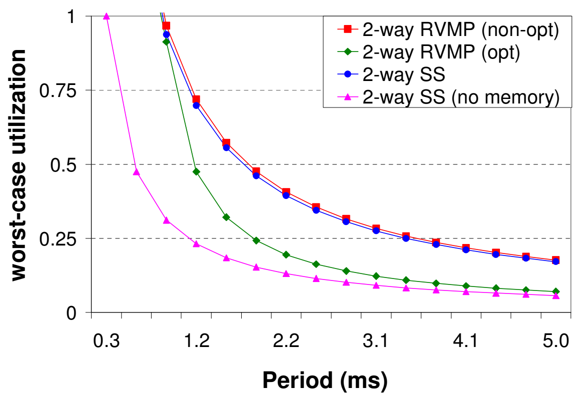


Figure 7-9. Worst-case utilization of FFT. RVMP: worst-case bus serialization ($n = 4$), no DRAM conflict ($s = 1$). SS: worst-case bus serialization ($n = 2$), no DRAM conflict ($s = 1$). $\frac{f_{proc}}{f_{bus}} = 2$.

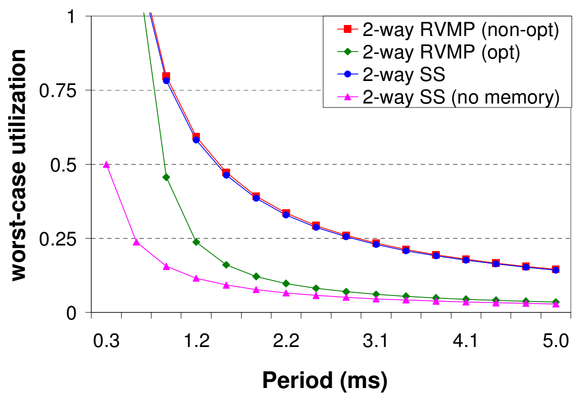
7.1.2.5 Effect of DRAM Conflicts on Schedulability

Figure 7-10 shows the effect of varying the degree of DRAM conflict ($s = 1, 2, 4$) on the worst-case utilization of FFT, assuming a fixed worst-case bus contention factor ($n = 4$).

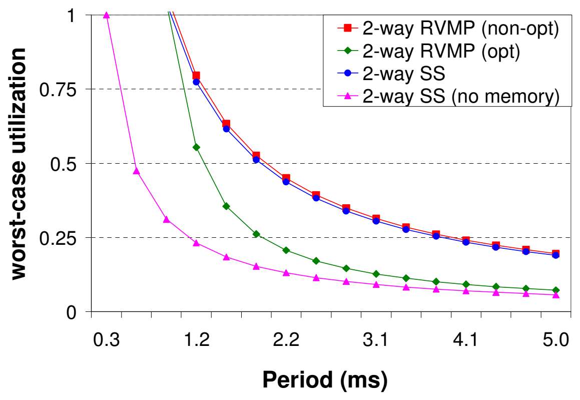
Notice that, the general trends observed in Section 7.1.2.4 for varying bus contention also apply here for varying DRAM conflicts. In general, as the memory component become a more dominant factor of WCET (either by increasing s or increasing $\frac{f_{proc}}{f_{bus}}$), the ability to overlap memory in an analyzable fashion becomes more valuable in improving the worst-case schedulability of the system. For example, in the worst-case memory contention scenario shown in Figure 7-10 ($n = 4$, $s = 4$, $\frac{f_{proc}}{f_{bus}} = 4$), the worst-case utilization gap between "RVMP (opt)" and "SS" reaches almost 40% at a period of 1.2 ms. In other words, using the tighter duty cycles accounting for memory overlap can reduce the worst-case utilization of the system in that case by 40%, which is a huge improvement in terms of worst-case performance.



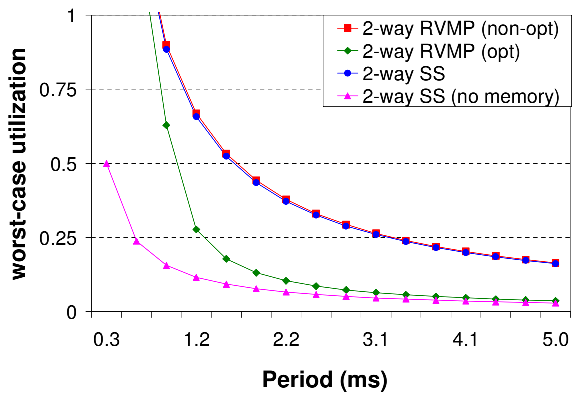
(a) $s = 1, \frac{f_{proc}}{f_{bus}} = 2$



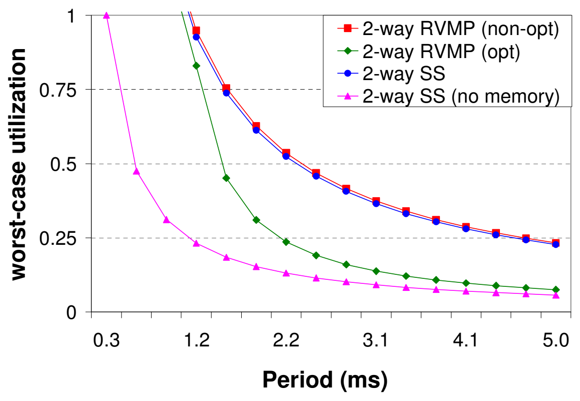
(b) $s = 1, \frac{f_{proc}}{f_{bus}} = 4$



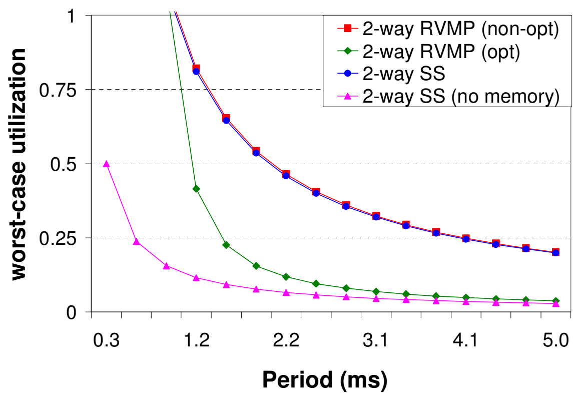
(c) $s = 2, \frac{f_{proc}}{f_{bus}} = 2$



(d) $s = 2, \frac{f_{proc}}{f_{bus}} = 4$



(e) $s = 4, \frac{f_{proc}}{f_{bus}} = 2$



(f) $s = 4, \frac{f_{proc}}{f_{bus}} = 4$

Figure 7-10. Worst-case utilization of FFT: worst-case bus contention ($n = 4$), varying DRAM conflicts ($s = 1, 2, 4$).

In Figure 7-11, we plot the worst-case utilization of FFT assuming both the worst-case bus serialization and worst-case DRAM conflicts. According to Section 7.1.1.2, this translates as $n = 4$ and $s = 4$ for the RVMP models and $n = 2$ and $s = 2$ for “2-way SS”. The advantage of reduced bus serialization and DRAM conflicts of “SS” over “RVMP (opt)” is clearly evident in Figure 7-11. For a considerable range of period ($period \leq 1.8$ ms), the worst-case utilization pf “SS” is better than that of “RVMP (opt)”, because of the difference in worst-case memory system contention. However, the ability of “RVMP (opt)” to overcome this penalty for $period > 1.8$ ms, where the worst-case utilization of “RVMP (opt)” becomes considerably better than that of “SS”.

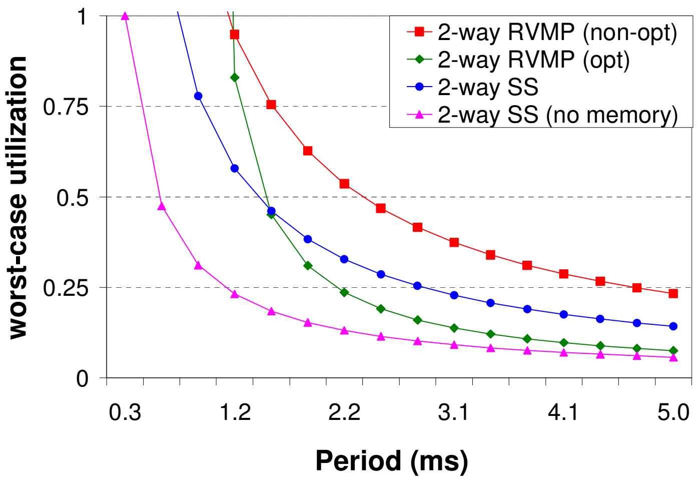


Figure 7-11. Worst-case utilization of FFT. RVMP: worst-case bus serialization ($n = 4$), worst-case DRAM conflict ($s = 4$). SS: worst-case bus serialization ($n = 2$), worst-case DRAM conflict ($s = 2$). $\frac{f_{proc}}{f_{bus}} = 2$.

7.2 Evaluation of Scalar RVMP with Memory Overlap

7.2.1 Methodology

We constructed various task-sets with different memory utilizations by combining tasks from Table 7-1. Each task-set in Table 7-3 is composed of a single task per virtual processor (assuming a four virtual processor system), and characterized by its memory-to-computation ratio. Task-sets with comparatively high, moderate, and low memory-to-computation ratios are referred to as HIGH, MED, and LOW, respectively. Task periods are chosen to yield a fully utilized system ($U = 1$) at 1 GHz using our proposed scalar RVMP substrate, *i.e.*, $\sum_i d_i \leq 1$. This implies the task-sets are just-feasible using our technique. Thus, if the task-set has a perceptible memory component, it will not be feasible using conventional EDF scheduling. This setup allows us to measure the over-subscription of the EDF schedule, whether or not task-sets will become feasible using EDF if processor frequency is increased, the amount of static slack achieved by RVMP over EDF, *etc.*

Table 7-3 lists the tasks in each task-set. The task's name, period (P), and individual utilization ($U_i = \frac{WCET_i}{P_i} = \frac{C_i+M_i+B_i}{P_i}$) are indicated for each task (WCET components were provided in Table 7-1). Notice that, since in the scalar RVMP case, there is no contention for function units, we use the unextended "non-RVMP" computation component of WCET (second column of Table 7-1). The second-to-last column of Table 7-3 gives the contribution of memory (DRAM + bus) to worst-case utilization of each task-set (assuming no overlap), revealing the memory-intensiveness of each task-set, ranging from 0.494 for HIGH down to 0.0470 for LOW. The last column gives the total worst-case utilization of each task-set using EDF scheduling - none of the task-sets are provably schedulable because their worst-case utilizations are greater than 1, failing the EDF schedulability test.

Similarly, task-sets composed of eight tasks each, two tasks per virtual processor are shown in Table 7-4.

Table 7-3. Task-sets characteristics
(4 tasks/task-set, scalar RVMP utilization = 1, $f_{proc} = 1$GHz, $f_{bus} = 500$MHz).

Task-set	**VP$_1$**			**VP$_2$**			**VP$_3$**			**VP$_4$**			**U$_{mem}$**	**U$_{total}$**
	Name	P (ms)	U	Name	P (ms)	U	Name	P (ms)	U	Name	P (ms)	U	(EDF)	(EDF)
LOW	adpcm	4.34	0.705	srt	14.9	0.172	lms	2.92	0.0700	crc	0.900	0.0660	0.0470	1.01
MED	mm	7.41	0.396	lame	29.0	0.337	fft	7.52	0.0920	toast	19.0	0.230	0.232	1.06
HIGH	cnt	0.375	0.316	cnt	0.375	0.316	cnt	0.446	0.265	cnt	0.446	0.265	0.494	1.16

Table 7-4. Task-sets characteristics
(8 tasks/task-set, scalar RVMP utilization = 1, $f_{proc} = 1$GHz, $f_{bus} = 500$MHz).

Task-set	**VP$_1$**			**VP$_2$**			**VP$_3$**			**VP$_4$**			**U$_{mem}$**	**U$_{total}$**
	Name	P (ms)	U	Name	P (ms)	U	Name	P (ms)	U	Name	P (ms)	U	(EDF)	(EDF)
LOW	adpcm	12.0	0.255	lms	7.00	0.0290	lame	50.0	0.196	srt	15.0	0.171	0.035	1.02
	srt	14.9	0.172	crc	9.00	0.0070	adpcm	20.0	0.153	lms	4.00	0.0510		
MED	mm	45.0	0.0650	fft	12.0	0.0580	cnt	2.50	0.0470	mm	1.00	0.293	0.25	1.04
	lame	30.0	0.362	toast	4.00	0.109	crc	2.00	0.0300	fft	6.00	0.115		
HIGH	cnt	0.700	0.169	cnt	0.800	0.149	cnt	1.00	0.118	cnt	0.375	0.315	0.480	1.10
	crc	0.850	0.0700	crc	0.600	0.0990	crc	0.900	0.0660	crc	0.500	0.119		

7.2.2 Results

In this section, we present worst-case schedulability experiments for baseline EDF (no overlap of tasks' WCETs) and our RVMP framework (overlap of tasks' WCETs). We show results for a single task per virtual processor and multiple tasks per virtual processor. In addition, we study the effect of varying the number of DRAM banks, including evaluating fewer banks than the number of virtual processors to understand the impact of limited DRAM parallelism. Finally, for demonstration, we simulate the baseline EDF and RVMP on the cycle-level simulator for 100 ms. In all cases, the simulation results are in agreement with the schedulability tests.

7.2.2.1 Schedulability Experiments

The graph in Figure 7-12 shows results of schedulability tests for a four virtual processor system with four DRAM banks. Each task-set has four tasks (thus, for RVMP, there is a single task per VP). The first bar ("EDF") is the worst-case utilization under EDF scheduling, *i.e.*, the sum of individual task utilizations, which must be less than 1 for schedulability. The next bar ("RVMP") is the worst-case utilization using our proposed RVMP infrastructure (taking into account overlapping memory accesses), *i.e.*, the sum of all tasks' duty cycles, which must be less than 1 for schedulability. Recall, task-sets are composed to achieve a worst-case utilization of 1 using RVMP at 1 GHz, and this is evident from the graph.

We also show a third bar, labeled "EDF (no memory)", which represents an ideal lower bound on worst-case utilization, achieved by ideally hiding and overlapping ALL the memory component of tasks. To model ideal overlap of computation and memory time, we set $M = 0$ and $B = 0$ (hiding all DRAM latency and bus transfer time) in the tasks' WCETs and plotted worst-case utilization accordingly. Thus, the difference between the "EDF" and "EDF (no memory)" bars is the memory component of worst-case utilization, including the bus transfer time (same as $\mathrm{U}_{\mathrm{mem}}$ column of Table 7-3). The larger this gap, the more potential reward for RVMP. This gap increases, going from least memory-intensive task-set (LOW) to most memory-intensive task-set (HIGH).

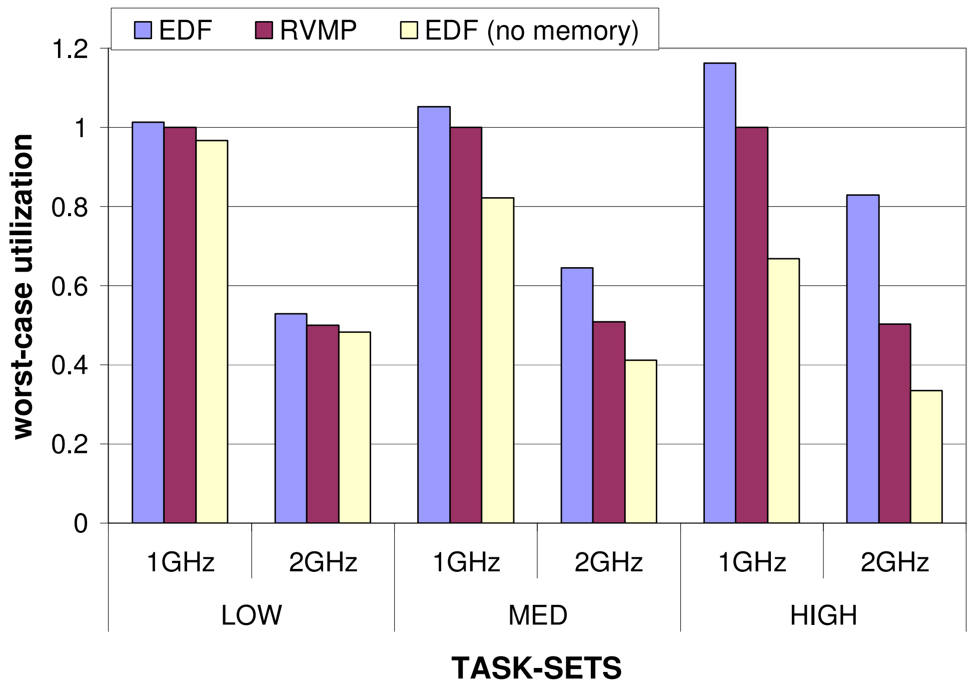


Figure 7-12. Worst-case utilization of scalar RVMP – task-sets with 4 tasks.

Task-sets LOW, MED, and HIGH are infeasible at 1 GHz using the conventional EDF (EDF worst-case utilization exceeds 1), whereas RVMP exploits overlapping WCETs in an analytically-bounded way to produce a feasible schedule.

Even for feasible EDF scenarios at the higher frequency (2 GHz), using RVMP results in more static slack in the schedule than does EDF, *e.g.*, Figure 7-12 shows around 50% slack for "RVMP" vs. only 18% for "EDF", for HIGH at 2 GHz. Static slack can be used to increase functionality via adding more tasks, reducing periods, *etc.* Notice that "RVMP" approaches the "EDF (no memory)" point, but does not perfectly overlap computation and memory time because memory transfer instructions initiate and complete in adjacent duty cycles, wasting an aggregate of one whole duty cycle during which the task could use the pipeline but does not. This is evident from the example memory transfer in Figure 6-1 (Section 6.1).

Figure 7-13 shows that our framework is scalable to systems where the number of tasks is greater than the number of available register contexts, by supporting multiple tasks per virtual processor. Conventional software EDF is used to schedule multiple tasks within a virtual processor. The same observations discussed previously, for a single task per VP, still apply for this more general case.

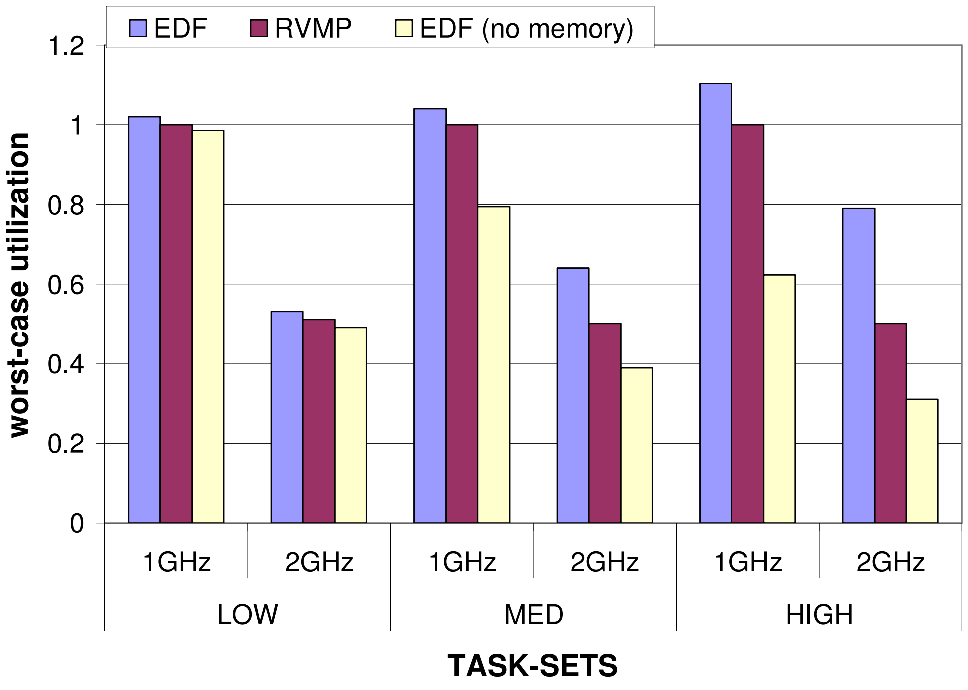


Figure 7-13. Worst-case utilization of scalar RVMP – task-sets with 8 tasks.

7.2.2.2 Effect of Varying DRAM Banks

In Figure 7-14, we show the effect of varying the number of DRAM banks (*i.e.*, DRAM parallelism) on the schedulability of RVMP. “EDF” and “EDF (no memory)” bars are the same as before, because they are not affected by the number of DRAM banks (no conflicts in the single processor case). “RVMP-1”, “RVMP-2”, and “RVMP-4” bars present scalar RVMP schedulability results for a four virtual processor system, with 1, 2, and 4 total DRAM banks, respectively. The trend is that schedulability improves with more banks, as anticipated. Somewhat surprisingly, note that "RVMP-1", which essentially serializes all memory accesses like "EDF", still performs better than "EDF" for the 2 GHz processors. Only at 1 GHz and 1 DRAM bank is the single-threaded EDF approach slightly preferred.

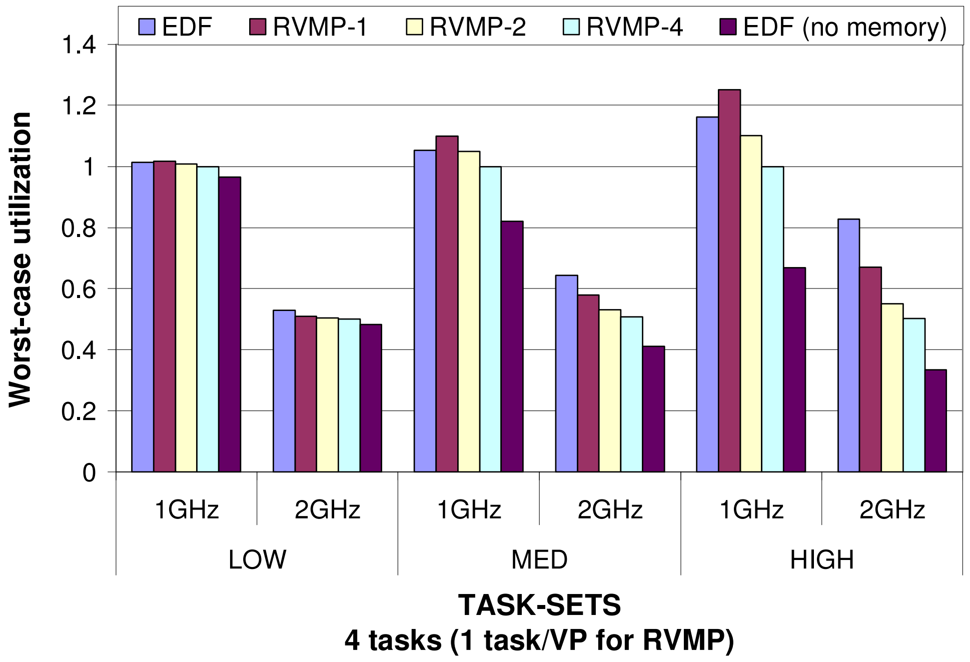


Figure 7-14. Varying the number of DRAM banks – scalar RVMP.

Although all memory accesses are essentially serialized in "RVMP-1" due to our very conservative assumptions regarding bank and bus conflicts, they are still overlapped with pipeline computation from other tasks, unlike "EDF". Results are very positive for HIGH at 2 GHz, a point that anticipates the memory wall in future embedded systems – notice that schedulability is universally good for RVMP with 1, 2, and 4 DRAM banks.

7.2.2.3 Simulation Demonstration

The graphs in Figure 7-15 show run-time utilization of task-sets with 4 tasks each, assuming parallel DRAM banks, measured by cycle-level simulation (Figure 7-16 shows similar results for task-sets with 8 tasks each). In addition to "EDF", "RVMP", and "EDF (no memory)" models described earlier, we show another bar, "EDF (switch-on-event)", which models a dynamic EDF scheduling policy augmented with switch-on-event (memory transfer) multithreading capability. When the highest priority task initiates a memory transfer, execution is dynamically switched to the next highest priority task.

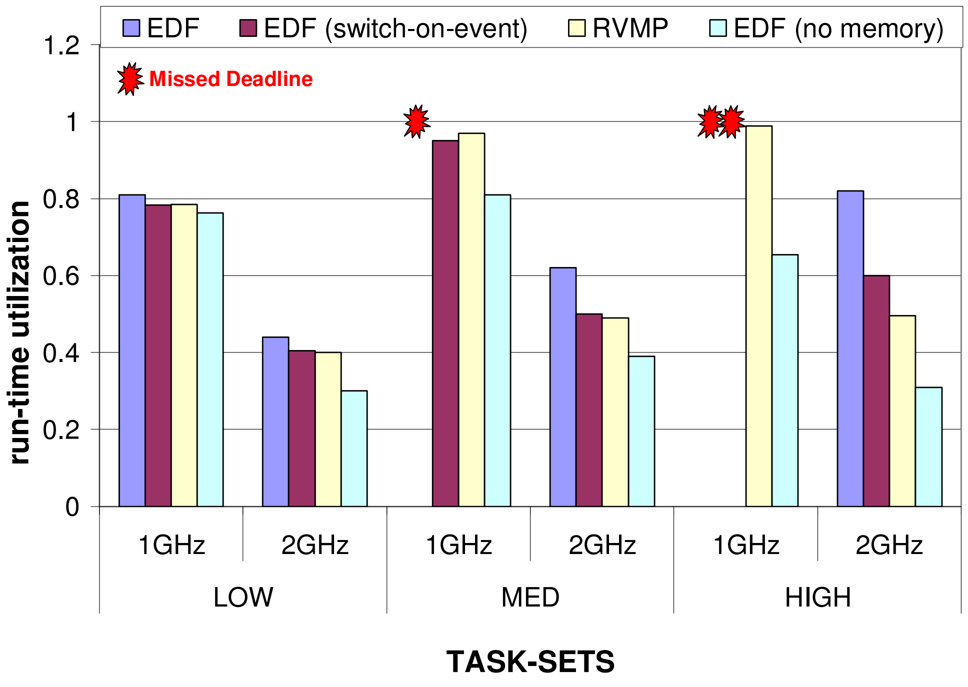


Figure 7-15. Run-time utilization of scalar RVMP – task-sets with 4 tasks.

Run-time utilization is naturally less than worst-case utilization, because it depends on actual execution times instead of worst-case execution times. All task-sets were successfully scheduled using RVMP at 1 GHz, with a run-time utilization less than 1 (as predicted by schedulability analysis). On the other hand, using EDF scheduling, task-sets HIGH and MED miss their deadlines and terminate as indicated on Figure 7-15 (explaining why the "EDF" bar is unavailable). Task-set LOW was successfully scheduled by EDF at 1 GHz due to the difference between WCETs and actual execution times. However, for hard-real-time system design, it is neither safe nor acceptable to use these actual execution times as inputs to worst-case schedulability tests.

Task-sets scheduled using "EDF (switch-on-event)" can unpredictably meet their deadlines (*e.g.*, LOW and MED at 1 GHz, and LOW, MED, and HIGH at 2 GHz) or miss their deadlines (*e.g.*, HIGH at 1 GHz). Moreover, in cases when task-sets were successfully scheduled, the run-time utilization of the task-sets could be better than "RVMP" in some cases (*e.g.*, MED at 1 GHz) or worse in other cases (*e.g.*, HIGH at 2 GHz), depending on the nature of the task-set itself and the location of memory transfers within the tasks. This re-emphasizes our initial observation regarding the dynamic nature of switch-on-event multithreading (Figure 1-7, Section 1.3). Although "EDF (switch-on-event)" can perform sometimes better than "RVMP", there is no reliable way to statically guarantee that performance, which is unsafe for hard-real-time systems.

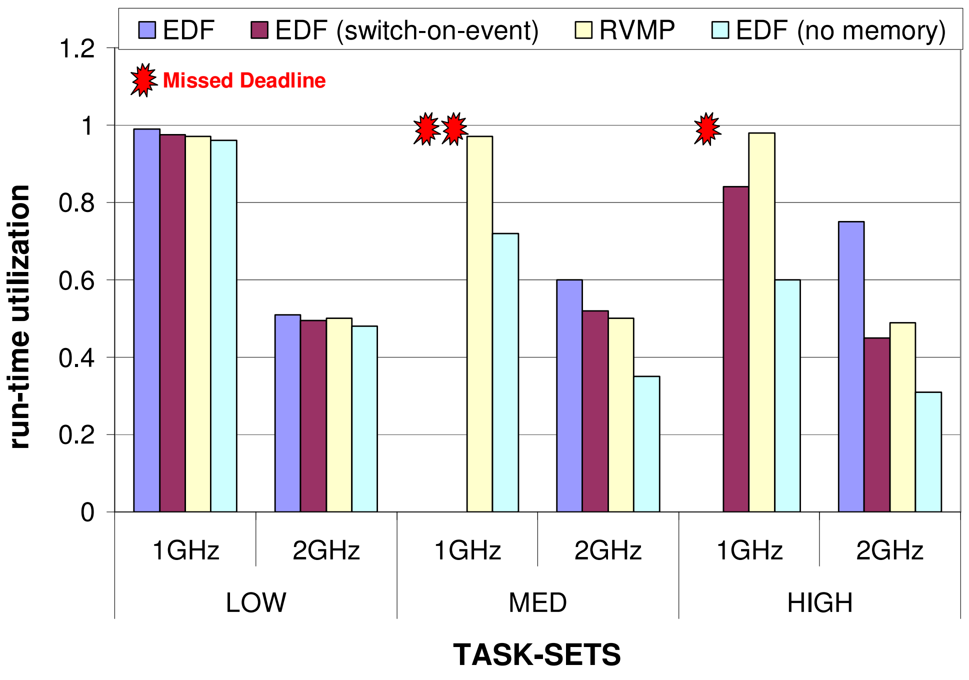


Figure 7-16. Run-time utilization of scalar RVMP – task-sets with 8 tasks.

Chapter 8

Summary

This dissertation proposes *flexible interference-free multithreading*, a novel microarchitectural approach that enables scheduling hard-real-time tasks on a simultaneous multithreading (SMT) substrate. The Real-time Virtual Multiprocessor (RVMP) framework consists of a flexible interference-free multithreading architecture and matching real-time scheduling analysis. RVMP virtualizes a single in-order superscalar processor into multiple interference-free different-sized virtual processors. The number and sizes of virtual processors can be rapidly reconfigured. RVMP's interference-free property preserves single-task WCET analysis, *i.e.*, tasks' WCETs can be derived independent of the task-set context. At the same time, RVMP's flexible space/time virtualization emulates fine-grain resource sharing of SMT to achieve similar cost-performance efficiency. A simple and effective real-time scheduling approach is proposed that concentrates scheduling into a small time interval, proving or disproving schedulability with very low complexity and at the same time producing a compact repeating space/time schedule that orchestrates processor virtualization.

RVMP successfully combines the analyzability (hence real-time formalism) of multiple processors with the flexibility (hence high performance) of simultaneous multithreading. Worst-case schedulability experiments show that RVMP is able to schedule more task-sets than rigid multiprocessor counterparts. We also observe that RVMP's advantage increases as task-sets become more demanding. We also confirm that multi-configuration schedules are crucial for schedulability, highlighting the need for rapid reconfiguration. Finally, RVMP is as effective as an SMT processor in run-time experiments, meanwhile providing a real-time formalism that SMT does not currently provide.

The base RVMP framework does not explicitly distinguish between computation and memory components of a task. Therefore, base RVMP assumes that the memory component is dilated like the computation component. In reality, the memory component is not dilated by the duty cycle, because memory accesses are handled by the memory system and are not pre-empted. By not accounting for this parallelism, memory overlap is underestimated in superscalar RVMP (note that superscalar RVMP implicitly models at least some degree of memory overlap, among currently active tasks on different partitions). Worse, in scalar RVMP, the naive memory model analytically yields no computation/memory overlap, since only one task is active at a time and WCET is monolithically treated.

We address this issue by distinguishing between computation and memory components of tasks, and exploiting the possible overlap between memory of tasks and computation of other tasks to calculate tighter safe duty cycles. The new memory overlap framework eliminates idle time that arises from over-estimating duty cycles, improving the worst-case performance of the system. In essence, RVMP's memory overlap framework captures the performance advantage of dynamic switch-on-event multithreading. Yet, it differs from switch-on-event multithreading in that it is analyzable. We derive a safe and tight bound on the overlap between memory accesses of tasks and computation of other tasks, irrespective of when the memory accesses actually happen within and among tasks. The memory overlap framework blends naturally with RVMP, feeding optimized duty cycles to RVMP's real-time scheduling analysis phase. In addition, a closed-form schedulability test is revealed for the scalar RVMP case, extending the classic EDF utilization test to account for memory overlap for the first time. The memory overlap framework also accounts for practical memory system issues, such as the degree of parallelism in the memory system (memory banks) and serialization on the bus.

This dissertation shows that taking a co-design approach to real-time systems, where processor architecture and real-time scheduling are designed together, can lead to significant improvement in hard-real-time performance. RVMP is an example of this approach. Our combined microarchitectural and scheduling approach makes it possible to achieve SMT-like performance with hard-real-time guarantees.

Chapter 9

Directions for Future Research

The contributions of this dissertation have significantly fleshed out and evaluated the RVMP paradigm. Nonetheless, there are many other interesting facets of RVMP that are not addressed here, and are left for future work. Some of these items are:

- Study the effect of function unit mix on the overall performance of RVMP. A specific time-shared function unit (such as the agen units or the FPU) may be a bottleneck, affecting schedulability of whole task-sets. If such a consistent "hot spot" is identified, the architecture can be modified to avoid it (*i.e.*, design the processor based on performance characterization).

- Modify the real-time scheduling analysis to handle contended function units more efficiently. This can be done by profiling the applications *a priori* to identify which function units are frequently or infrequently used by specific tasks. The guaranteed percentage of function unit bandwidth can then be assigned based on this profile information instead of the width of the partition. A similar approach can also be applied to handle contended bus bandwidth more efficiently.

- Explore the possibility of improving the efficiency of the RVMP scheduling framework by narrowing down the search space of bin-packing. That is, we want to try to come up with more tests for discarding infeasible architectural configurations early without explicit bin-packing (similar to the total available packing area test we introduce in Section 4.1). Better yet, it is desirable to explore mathematical frameworks that compute (solve) space/time schedules directly, avoiding bin-packing searches altogether. Improving the scheduler efficiency is especially useful for online scheduling.

- Improve the bound on the expansion of the bus component of WCET due to serialization on the bus. The conservatively expanded bus time ($n \times B$), discussed in Section 6.3.1, is the major performance limiter of the safe memory overlap technique. Improving this bound will greatly improve the hard-real-time performance (*i.e.*, less conservative schedulability analysis) of the system. This will also be valuable for traditional multiprocessor real-time systems with shared memory buses.

- Explore using RVMP for other hard-real-time task models, such as *sporadic* real-time tasks. RVMP provides a nice infrastructure for developing an online acceptance test for sporadic tasks. When a sporadic task is released, the acceptance test tries to fit it within the current space/time round schedule, based on the sporadic task's required duty cycles on partially idle partitions. There may be static idle time (spare capacity in worst-case round schedule) and/or dynamic slack (early task completion). The test might decide to re-partition the processor on-the-fly in order to fit the sporadic task, *i.e.*, redo bin-packing from scratch on-the-fly.

- Design an RVMP-compatible dynamic voltage scaling (DVS) approach to reduce energy consumption. DVS on RVMP is challenging. Any DVS approach will have to account for the fact that all VPs are affected by the frequency scaling. In contrast, on a conventional multiprocessor, DVS can flexibly adjust frequencies of different processors independently.

- Evaluate the potential of using RVMP for soft-real-time and non-real-time systems, as a measure of providing guaranteed performance levels (or quality-of-service) for applications in an SMT environment. RVMP provides stronger guarantees for tasks than past work on soft-real-time SMT [49].

Bibliography

[1] A. Agarwal. Performance Tradeoffs in Multithreaded Processors. *IEEE Trans. on Parallel and Distributed Systems*, 3(5):525–539, 1992.

[2] H. Akkary and M. A. Driscoll. A Dynamic Multithreading Processor. In *Proc. of the 31st Int'l Symp. on Microarchitecture*, pages 226–236, Dec. 1998.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proc. of the 4th Int'l Conf. on Supercomputing*, pages 1–6, June 1990.

[4] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems. In *Proc. of the 30th Int'l Symp. on Computer Architecture*, pages 350–361, June 2003.

[5] A. Anantaraman, K. Seth, E. Rotenberg, and F. Mueller. Enforcing Safety of Real-Time Schedules on Contemporary Processors Using a Virtual Simple Architecture (VISA). In *Proc. of the 25th Int'l Real-Time Systems Symp.*, pages 114–125, Dec. 2004.

[6] A. Anantaraman, K. Seth, E. Rotenberg, and F. Mueller. Exploiting VISA for Higher Concurrency in Safe Real-Time Systems. Tech. Report TR-2004-15, CS Dept., NC State Univ., May 2004.

[7] R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding Worst-Case Instruction Cache Performance. In *Proc. of the 15th Int'l Real-Time Systems Symp.*, pages 172–181, Dec. 1994.

[8] T. P. Baker and A. C. Shaw. The Cyclic Executive Model and Ada. In *Proc. of the 9th Int'l Real-Time Systems Symp.*, pages 120–129, Dec. 1988.

[9] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: Design Alternative for Cache On-Chip Memory in Embedded Systems. In *Proc. of the 10th Int'l Symp. on Hardware/Software Codesign*, pages 73–78, May 2002.

[10] I. Bate, P. Conmy, T. Kelly, and J. A. McDermid. Use of Modern Processors in Safety-Critical Applications. *The Computer Journal*, 44(6):531–543, 2001.

[11] F. Bodin and I. Puaut. A WCET-Oriented Static Branch Prediction Scheme for Real-Time Systems. In *Proc. of the 17th Euromicro Conf. on Real-Time Systems*, July 2005.

[12] U. Brinkschulte, J. Kreuzinger, M. Pfeffer, and T. Ungerer. A Scheduling Technique Providing a Strict Isolation of Real-time Threads. In *Proc. of the 7th IEEE Int'l Workshop on Object-Oriented Real-Time Dependable Systems*, pages 334–342, Jan. 2004.

[13] D. Burger, T. M. Austin, and S. Bennett. The Simplescalar Toolset, Version 2. Tech. Report CS-TR-1997-1342, CS Dept., Univ. of Wisconsin-Madison, July 1997.

[14] A. Burns. Scheduling Hard Real-Time Systems: A Review. *Software Engineering Journal*, 6(3):116–128, 1991.

[15] J. Burns and J.-L. Gaudiot. SMT Layout Overhead and Scalability. *IEEE Trans. on*

Parallel and Distributed Systems, 13(2):142–155, 2002.

[16] C-Lab. WCET Benchmarks. Available from http://www.c-lab.de.

[17] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramírez, and M. Valero. Predictable Performance in SMT Processors. In *Proc. of the 2004 Conf. on Computing Frontiers*, pages 433–443, Apr. 2004.

[18] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramírez, and M. Valero. Architectural Support for Real-Time Task Scheduling in SMT Processors. In *Proc. of the 2005 Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 166–176, Sept. 2005.

[19] F. J. Cazorla, A. Ramírez, M. Valero, and E. Fernández. Dynamically Controlled Resource Allocation in SMT Processors. In *Proc. of the 37th Int'l Symp. on Microarchitecture*, pages 171–182, Nov. 2004.

[20] F. J. Cazorla, A. Ramírez, M. Valero, P. M. W. Knijnenburg, R. Sakellariou, and E. Fernández. QoS for High-Performance SMT Processors in Embedded Systems. *IEEE Micro*, 24(4):24–31, 2004.

[21] B. Chazelle. The Bottom-Left Bin-Packing Heuristic: An Efficient Implementation. *IEEE Trans. on Computers*, 32(8):697–707, 1983.

[22] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *Proc. of the 22nd Int'l Symp. on Computer Architecture*, pages 333–344, June 1995.

[23] D. Cormie. The ARM11 Microarchitecture. White paper, Apr. 2002.

[24] P. Crowley and J.-L. Baer. Worst-Case Execution Time Estimation of Hardware-assisted Multithreaded Processors. In *Proc. of the 2nd Workshop on Network Processors*, pages 36–47, Feb. 2003.

[25] A. G. Dean and J. P. Shen. Techniques for Software Thread Integration in Real-Time Embedded Systems. In *Proc. of the 19th Int'l Real-Time Systems Symp.*, pages 322–333, Dec. 1998.

[26] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1):127–140, 1978.

[27] D. Donalson, M. Serrano, R. Wood, and M. Nemirovsky. DISC: Dynamic Instruction Stream Computer: An Evaluation of Performance. In *Proc. of the 26 Hawaii Int'l Conf. on system Science*, pages 448–456, Jan. 1993.

[28] J. E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation Algorithms for Bin Packing: A Survey. In *Approximation Algorithms for $\mathcal{NP}$-Hard Problems*, pages 46–93. PWS Publishing Co., 1997.

[29] J. H. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. *IEEE Micro*, 15(2):33–43, 1995.

[30] R. J. Eickemeyer, R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu. Evaluation of Multithreaded Uniprocessors for Commercial Application Environments. In *Proc. of the 23rd Int'l Symp. on Computer Architecture*, pages 203–212, May 1996.

[31] A. El-Haj-Mahmoud, A. S. AL-Zawawi, A. Anantaraman, and E. Rotenberg. Virtual Multiprocessor: An Analyzable, High-Performance Microarchitecture for Real-Time Computing. In *Proc. of the 2005 Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 213–224, Sept. 2005.

[32] A. El-Haj-Mahmoud and E. Rotenberg. Safely Exploiting Multithreaded Processors to Tolerate Memory Latency in Real-Time Systems. In *Proc. of the 2004 Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 2–13, Sept. 2004.

[33] A. El-Moursy and D. H. Albonesi. Front-End Policies for Improved Issue Efficiency in SMT Processors. In *Proc. of the 9th Int'l Symp. on High Performance Computer Architecture*, pages 31–42, Feb. 2003.

[34] J. Engblom. On Hardware and Hardware Models for Embedded Real-Time Systems. In *Embedded Real-Time Systems Workshop*, Dec. 2001.

[35] A. Falcón, A. Ramírez, and M. Valero. A Low-Complexity, High-Performance Fetch Unit for Simultaneous Multithreading Processors. In *Proc. of the 10th Int'l Symp. on High Performance Computer Architecture*, pages 244–253, Feb. 2004.

[36] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-Machine Multicomputer. In *Proc. of the 28th Int'l Symp. on Microarchitecture*, pages 146–156, Nov. 1995.

[37] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: A Compiler Framework for Analyzing and Tuning Memory Behavior. *ACM Trans. on Programming Languages and Systems*, 21(4):703–746, 1999.

[38] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. of the 4th Workshop on Workload Characterization*, Dec. 2001.

[39] T. Hand. Real-Time Systems Need Predictability. *Computer Design (RISC Supplement)*, pages 57–59, Aug. 1989.

[40] M. G. Harmon, T. P. Baker, and D. B. Whalley. A Retargetable Technique for Predicting Execution Time of Code Segments. In *Proc. of the 13th Int'l Real-Time Systems Symp.*, pages 68–77, Dec. 1992.

[41] A. Hartstein and T. R. Puzak. Optimum Power/Performance Pipeline Depth. In *Proc. of the 36th Int'l Symp. on Microarchitecture*, page 117, Dec. 2003.

[42] C. A. Healy, R. D. Arnold, F. Mueller, M. G. Harmon, and D. B. Walley. Bounding Pipeline and Instruction Cache Performance. *IEEE Trans. on Computers*, 48(1):53–70, 1999.

[43] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proc. of the 16th Int'l Real-Time Systems Symp.*, pages 288–297, Dec. 1995.

[44] S. Hily and A. Seznec. Out-of-Order Execution may not be Cost-Effective on Processors Featuring Simultaneous Multithreading. In *Proc. of the 5th Int'l Conf. on High Performance Computer Architecture*, page 64, Jan. 1999.

[45] IBM Corp. IBM PowerPC 740 / PowerPC 750 RISC Microprocessor User's Manual. Feb. 1999.

[46] Intel Corp. Intel IXP2850 Network Processor. Available from
http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm.

[47] B. Iyer, S. Srinivasan, and B. L. Jacob. Extended Split-Issue: Enabling Flexibility in the Hardware Implementation of NUAL VLIW DSPs. In *Proc. of the 31st Int'l Symp. on Computer Architecture*, pages 364–375, June 2004.

[48] L. E. Jackson and G. N. Rouskas. Deterministic Preemptive Scheduling of Real-Time Tasks. *IEEE Computer*, 35(5):72–79, 2002.

[49] R. Jain, C. J. Hughes, and S. V. Adve. Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. In *Proc. of the 23rd Int'l Real-Time Systems Symp.*, pages 134–145, Dec. 2002.

[50] S. Kaxiras, G. J. Narlikar, A. D. Berenbaum, and Z. Hu. Comparing Power Consumption of an SMT and a CMP DSP for Mobile Phone Workloads. In *Proc. of the 2001 Int'l Conf. on Compilers, Arch., and Syn. for Embedded Systems*, pages 211–220, Nov. 2001.

[51] S. W. Keckler and W. J. Dally. Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism. In *Proc. of the 19th Int'l Symp. on Computer Architecture*, pages 202–213, June 1992.

[52] S.-K. Kim, S. L. Min, and R. Ha. Efficient Worst Case Timing Analysis of Data Caching. In *Proc. of the 2nd Real Time Technology and Applications Symp.*, pages 230–240, 1996.

[53] D. B. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In *Proc. of the 10th Int'l Real-Time Systems Symp.*, pages 229–239, Dec. 1989.

[54] J. Kreuzinger, A. Schulz, M. Pfeffer, and T. Ungerer. Real-Time Scheduling on Multithreaded Processors. In *Proc. of the 7th Workshop on Real-Time Computing and Applications Symp.*, pages 155–159, Dec. 2000.

[55] V. Krishnan and J. Torrellas. Efficient Use of Processing Transistors for Larger On-Chip Storage: Multithreading. In *Proc. of Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, July 1997.

[56] S. Lauzac, R. Melhem, and D. Mosse. Comparison of Global and Partitioning Schemes for Scheduling Rate Monotonic Tasks on a Multiprocessor. In *Proc. of the 10th Euromicro Workshop on Real-Time Systems.*, page 188, 1998.

[57] J. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Systems. *Performance Evaluation*, 2(1):37–250, 1982.

[58] X. Li, A. Roychoudhury, and T. Mitra. Modeling Out-of-Order Processors for Software Timing Analysis. In *Proc. of the 25th Int'l Real-Time Systems Symp.*, pages 92–103, Dec. 2004.

[59] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proc. of the 16th Int'l Real-Time Systems Symp.*, pages 298–307, Dec. 1995.

[60] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son. New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems. *IEEE Trans. on Computers*, 44(12):1429–1442, 1995.

[61] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In *Proc. of the 19th Int'l Real-Time Systems Symp.*, pages

334–345, Dec. 1998.

[62] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[63] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[64] T. Lundqvist and P. Stenström. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *Real-Time Systems*, 17(2-3):183–207, 1999.

[65] T. Lundqvist and P. Stenström. Empirical Bounds on Data Caching in High-Performance Real-Time Systems. Tech. report, Chalmers Univ. of Technology, 1999.

[66] B. I. Moon, M. G. Kim, I. P. Hong, K. C. Kim, , and Y. S. Lee. Study of an In-order SMT Architecture and Grouping Schemes. *Int'l Journal of Control, Automation, and Systems*, 1(3):339–350, 2003.

[67] F. Mueller. Compiler Support for Software-Based Cache Partitioning. In *Proc. of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 125–133, 1995.

[68] M. D. Nemirovsky, F. Brewer, and R. C. Wood. DISC: Dynamic Instruction Stream Computer. In *Proc. of the 24th Int'l Symp. on Microarchitecture*, pages 163–171, Nov. 1991.

[69] Y. Oh and S. H. Son. Fixed Priority Scheduling of Periodic Tasks on Multiprocessor Systems. CS Tech. Report, Univ. of Virginia, Mar. 1995.

[70] E. Özer and T. M. Conte. High-Performance and Low-Cost Dual-Thread VLIW Processor

Using Weld Architecture Paradigm. *IEEE Trans. on Parallel and Distributed Systems*, 16(12):1132–1142, 2005.

[71] E. Özer, T. M. Conte, and S. Sharma. Weld: A Multithreading Technique Towards Latency-Tolerant VLIW Processors. In *Proc. of the 8th Int'l Conf. on High Performance Computing*, pages 192–203, Dec. 2001.

[72] P. R. Panda, N. D. Dutt, and A. Nicolau. On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems. *ACM Trans. on Design Automation of Electronic Systems*, 5(3):682–704, 2000.

[73] S. Parekh, S. Eggers, and H. Levy. Thread-Sensitive Scheduling of SMT Processors. Tech. report, Dept. of CS and Eng'g, Univ. of Washington, 2000.

[74] S. Raasch and S. Reinhardt. The Impact of Resource Partitioning on SMT Processors. In *Proc. of the 12th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 15–26, Sept. 2003.

[75] A. Raha, N. Malcolm, and W. Zhao. Hard Real-Time Communications with Weighted Round Robin Service in ATM Local Area Networks. In *Proc. of the 1st Int'l Conf. on Engineering of Complex Computer Systems*, page 96, 1995.

[76] H. Ramaprasad and F. Mueller. Bounding Worst-Case Data Cache Behavior by Analytically Deriving Cache Reference Patterns. In *Proc. of the 11th Int'l Real-Time and Embedded Technology and Applications Symp.*, pages 148–157, Mar. 2005.

[77] E. Rotenberg. Using Variable-MHz Microprocessors to Efficiently Handle Uncertainty in Real-Time Systems. In *Proc. of the 34th Int'l Symp. on Microarchitecture*, pages 28–39,

Dec. 2001.

[78] V. Sardana. The MIPS32 34K Core Family: Powering Next-Generation Embedded SoCs. White paper, Feb. 2006.

[79] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. FAST: Frequency-Aware Static Timing Analysis. In *Proc. of the 24th Int'l Real-Time Systems Symp.*, pages 40–51, Dec. 2003.

[80] B. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Proc. of the 4th Symp. on Real Time Signal Processing IV*, pages 241–248, 1981.

[81] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd Int'l Symp. on Computer Architecture*, pages 414–425, June 1995.

[82] G. S. Sohi and M. Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Proc. of the 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, Apr. 1991.

[83] A. Srinivasan, P. Holman, J. H. Anderson, and S. Baruah. The Case for Fair Multiprocessor Scheduling. In *Proc. of the 17th Int'l Symp. on Parallel and Distributed Processing*, page 114.1, Apr. 2003.

[84] J. A. Stankovic, I. Lee, A. K. Mok, and R. Rajkumar. Opportunities and Obligations for Physical Computing Systems. *IEEE Computer*, 38(11):23–31, Nov. 2005.

[85] J. A. Stankovic, K. Ramamritham, and M. Spuri. *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms*. Kluwer Academic Publishers, 1998.

[86] J. Stohr, A. von Bülow, and G. Färber. Bounding Worst-Case Access Times in Modern Multiprocessor Systems. In *Proc. of the 17th Euromicro Conf. on Real-Time Systems*, pages 189–198, July 2005.

[87] R. Thekkath and S. J. Eggers. The Effectiveness of Multiple Hardware Contexts. In *Proc. of the 6th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 328–337, Oct. 1994.

[88] N. Tuck and D. M. Tullsen. Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. In *Proc. of the 12th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 26–35, Sept. 2003.

[89] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of the 23rd Int'l Symp. on Computer Architecture*, pages 191–202, May 1996.

[90] Ubicom, Inc. The Ubicom IP3023 Wireless Network Processor. White paper, Apr. 2003.

[91] S. Udayakumaran and R. Barua. Compiler-Decided Dynamic Memory Allocation for Scratch-pad Based Embedded Systems. In *Proc. of the 2003 Int'l Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, pages 276–286, Sept. 2003.

[92] T. Ungerer, B. Robic, and J. Silc. A Survey of Processors with Explicit Multithreading. *ACM Computing Surveys*, 35(1):29–63, 2003.

[93] X. Vera and J. Xue. Let's Study Whole-Program Cache Behaviour Analytically. In *Proc. of the 8th Int'l Symp. on High Performance Computer Architecture*, pages 175–186, Feb. 2002.

[94] B. J. Welch, S. O. Kanaujia, A. Seetharam, D. Thirumalai, and A. G. Dean. Supporting Demanding Hard-Real-Time Systems with STI. *IEEE Trans. on Computers*, 54(10):1188–1202, 2005.

[95] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. In *Proc. of the 3rd Real-Time Technology and Applications Symp.*, page 192, 1997.

[96] A. Wolfe. Software-Based Cache Partitioning for Real-Time Applications. *Computer Software Engineering*, 2(3):315–327, 1994.

[97] A. Wolfe and J. P. Shen. A Variable Instruction Stream Extension to the VLIW Architecture. In *Proc. of the 4th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 2–14, Apr. 1991.

[98] F. Xie, M. Martonosi, and S. Malik. Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits. In *Proc. of the 2003 Conf. on Programming Language Design and Implementation*, pages 49–62, June 2003.

[99] W. Yamamoto and M. Nemirovsky. Increasing Superscalar Performance through Multistreaming. In *Proc. of the 4th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.

Appendices

Appendix A

Calculating the Size of the HRT

In this appendix, we derive an equation to calculate the storage size of the HRT for a general RVMP processor (*i.e.*, with configurations other than that presented in this dissertation).

To begin, let's first define the following variables:

- w: width of superscalar processor (*i.e.*, number of superscalar ways).
- v: number of virtual processors, which is equal to the number of hardware threads.
- k: number of function units or execution pipelines.

Since there can be a maximum of v reconfigurations per round (pure time-sharing case), the maximum number of HRT entries required is also v. Each entry consists of the following:

- A lifetime counter (LTC). The number of bits of the LTC should be enough to represent the longest possible round, referred to as R_MAX. Thus, the number of required bits for the LTC is $log_2(R\_MAX)$.

- A fetch vector (FV). The FV is an w-cycle cyclic schedule for the fetch unit. Each entry of the FV represents a virtual processor to fetch from, requiring $log_2(v)$ bits. The total size of the FV is $w \times log_2(v)$ bits.

- A partitioning vector (PV). The PV is an w-entry vector, where each entry indicates a virtual processor. Similar to the FV, the size of the PV is $w \times log_2(v)$ bits.

- k configuration vectors (CVs), one for each function unit. Like the FV, the size of each CV is $w \times log_2(v)$ bits. The total storage size of all CVs is $k \times n \times log_2(v)$ bits.

- A cycle count (CC). The CC is used to index the FV and all k CVs. Thus, the total number of bits required by the CC is $log_2(\text{number of entries of FV and CVs}) = log_2(w)$ bits.

- A 1-bit end-of-table (EOT) flag.

The total size of the HRT (in bits) is the total size of each entry times the number of entries:

$$\begin{aligned}\text{HRT size } = v\times\Big(&log_2(R\_MAX) + w \times log_2(v) + w \times log_2(v) \\ &+ k \times w \times log_2(v) + log_2(w) + 1\Big)\end{aligned}$$

Rearranging this expression, we can write the HRT size equation as follows:

Equation A-1.

$$\text{HRT size } = v \times \Big(log_2(R\_MAX) + (2+k) \times w \times log_2(v) + log_2(w) + 1\Big)$$

Using Equation A-1, we can calculate the size of the HRT in Section 3.4 by substituting the following values: width of superscalar processor $w = 4$, number of virtual processors $v = 4$, number of function units $k = 5$, and maximum round length $R\_MAX = 4096$.

$$\text{HRT size } = 4 \times \Big(log_2(4096) + (2+5) \times 4 \times log_2(4) + log_2(4) + 1\Big) = 284 \text{ bits} = 35.5 \text{ bytes}$$

Appendix B

Detailed Derivation of the Analytical Model for Memory Overlap

Each task's WCET is divided into computation (C) and memory (M) components. The number of *whole* rounds (*i.e.*, assuming no disruptions by memory transfers in the middle of duty cycles) needed to complete the computation component of a task is its computation time divided by the time per round allocated to the task, or

$$N = \left\lceil \frac{C}{d \times R} \right\rceil$$

where C is aggregate computation time, d is the duty cycle, R is the round time, and N is the number of whole rounds. This expression holds in spite of disruptions by memory transfers and is independent of when these disruptions occur. When computation is disrupted during a duty cycle by a memory transfer, computation resumes at the corresponding point in the next duty cycle, as explained in Section 6.1. Since we separate out memory time explicitly, the effect is to concatenate complementary computation portions of adjacent duty cycles, as if the disruptions had not occurred.

The time needed to finish the computation component is the number of whole rounds multiplied by the round time, or

$$N \times R = \left\lceil \frac{C}{d \times R} \right\rceil \times R$$

Since individual memory transfers always begin and end in consecutive duty cycles, we ensure that there is no idle time following transfers. Therefore, aggregate memory time M is not dilated. Thus, we get the following expression for WCET', the dilated WCET:

$$WCET' = \left(\left\lceil \frac{C}{d \times R} \right\rceil \times R \right) + M$$

The ceiling function $\lceil \quad \rceil$ is a necessary precaution. An interval of time equal to the round is guaranteed to contain one full duty cycle (in aggregate), regardless of where the interval starts and ends. The ceiling function produces an integer number of rounds, N, guaranteeing N duty cycles regardless of where the task is released.

In order for the task to safely meet its deadline, the dilated WCET (WCET') must be less than the task's deadline (which is the period in this case):

Equation B-1. $$\left(\left\lceil \frac{C}{d \times R} \right\rceil \times R \right) + M \leq period$$

It turns out that, if we constrain the period to be an integer multiple of the round, then we can correctly remove the ceiling function from the left-hand side of Equation B-1, which is confirmed later in this section. We do not sacrifice system timing specifications if we replace the period in Equation B-1 with a tighter period that is an integer multiple of the round:

$$period' = \left\lfloor \frac{period}{R} \right\rfloor \times R$$

If we remove the ceiling function from Equation B-1, replace $period$ with $period'$ (tighter constraint), and solve for d, we get:

$$d \leq \frac{C}{period' - M}$$

Solving for the lower limit of d (to minimize overall utilization) we get:

Equation B-2.
$$d = \frac{C}{period' - M}$$

Notice that Equation B-2 is the same as Equation 6-2 we intuitively derived in Section 6.1, but with $period'$ instead of $period$.

We now substitute this d back into Equation B-1 to confirm that initially removing the ceiling function is correct, assuming the modified period. This yields the following:

$$\left\lceil \frac{period' - M}{R} \right\rceil \leq \frac{period' - M}{R}$$

This condition only holds if both $period'$ and M are integer multiples of R, which is the case: (1) the round R is equal to the memory latency, and M is an integer multiple of the memory latency; (2) we defined $period'$ to be an integer multiple of the round R.

The impact of using $period\prime$ versus $period$ is minor because the round is typically a small fraction of the period. For a processor running at 1GHz, the round is usually of the order of 100 of cycles, while task periods in a typical real-time system are of the order of 100,000 of cycles. Thus, for all practical purposes, we can assume that $\frac{period}{period'} \approx 1$.

A more significant effect (but still relatively small) is rounding up $d \times R$ (*i.e.*, the duration of the duty cycle) to be an integer number of cycles of the round, during which the task is active. Duty cycle rounding is only a problem for a task-set that is barely feasible using conventional EDF scheduling and that does not have a perceptible memory component. With no memory to overlap, duty cycle rounding is enough to make the task-set barely infeasible using RVMP. In this case, we can simply revert to using conventional EDF scheduling.

Appendix C

Memory-Overlap vs. Non-Memory-Overlap Duty Cycles

When duty cycles are computed, the basic RVMP scheme does not take into consideration the possibility of overlapping memory accesses from one task with computation from another. However, the analytical model presented in Chapter 6 accounts for that overlap. In this section, we provide a mathematical proof that accounting for memory overlap provides a better overall worst-case utilization.

We refer to the duty cycle calculated with out taking memory overlap into consideration as the *unoptimized* duty cycle ($d_{non-opt}$). Note that this is also the EDF utilization of the task. Recall from Section 4.1.1 that:

$$d_{non-opt} = U = \frac{WCET}{period} = \frac{C + M}{period}$$

Note that since the basic scheme doesn't distinguish between C and M components, the WCET of a task is calculated as simply the sum of both component (*i.e.*, this assumes that M is dilated by the duty cycle, just like C).

On the other hand, the *optimized* duty cycle (d_{opt}) that considers memory overlap can be calculated as discussed in Section 6.1:

$$d_{opt} = \frac{C}{period - M}$$

For a given task to be feasible, its duty cycle must be less than or equal to 1, which results in the following condition for both the optimized and unoptimized duty cycles:

$$\begin{aligned} d_{non-opt} \leq 1 \quad &\Rightarrow \quad period \geq C + M \\ d_{opt} \leq 1 \quad &\Rightarrow \quad period - M \geq C \Rightarrow period \geq C + M \end{aligned}$$

In order to maximize the worst-case utilization of the whole system, we'd like to minimize the duty cycle needed to schedule any individual task. In other words, the smaller of d_{opt} and $d_{non-opt}$ will yield a better overall worst-case utilization. the minimum of d_{opt} and $d_{non-opt}$. To find the smaller of the two duty cycles, we subtract both expressions:

$$
\begin{aligned}
d_{non-opt} - d_{opt} &= \frac{C+M}{period} - \frac{C}{period - M} \\
&= \frac{(C+M)(period - M) - period \times C}{period(period - M)} \\
&= \frac{period \times C + M \times period - M \times C - M^2 - period \times C}{period(period - M)} \\
&= \frac{M(period - C - M)}{period(period - M)}
\end{aligned}
$$

Since $period \geq C + M$ (the condition necessary for any single task to be feasible) and $period$, C, and M are all positive numbers, we can conclude that both the numerator and dominator of the above expression are positive quantities. This means that $d_{non-opt} - d_{opt} \geq 0$. In other words, choosing the optimized duty cycle will always minimize the utilization required by a single task, thus, maximize the system's worst-case utilization. This is true for all cases that are covered by the feasibility condition above, which are the cases we care about anyway.

Notice that since both RVMP models, with and without memory overlap technique, suffer from the same constraints regarding bus contention and DRAM conflicts, the above derivation is still valid in the more general case. The memory component, M, can be replaced with the general expression accounting for conflicts: $s \times M + n \times B$.