HIGH-PERFORMANCE COMPUTER SYSTEM ARCHITECTURES

FOR EMBEDDED COMPUTING

A Thesis
Presented to
The Academic Faculty

by

Dongwon Lee

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2011

HIGH-PERFORMANCE COMPUTER SYSTEM ARCHITECTURES

FOR EMBEDDED COMPUTING

Approved by:

Dr. Marilyn Wolf, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Saibal Mukhopadhyay
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Shuvra Bhattacharyya
Department of Electrical and Computer Engineering
University of Maryland, College Park

Dr. David Anderson
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: August 24, 2011

To my parents,

and

my wife,
my daughter.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor Professor Marilyn Wolf for her guidance throughout my Ph.D. study at Georgia Tech. Dr. Wolf introduced me to research in embedded computing system architectures. She gradually challenged me with interesting research topics and gave me the extensive opportunity to study various areas in embedded computing system architectures.

I would like to thank Professor Shuvra Bhattacharyya of University of Maryland at College Park for advising me on research in graph representations of DSP applications. I also would like to thank Professor Hyesoon Kim for providing me with the opportunity to work at GPU architectures.

I am grateful to my proposal and defense committee members, Professor Sudhakar Yalamanchili, Professor David Anderson, and Professor Saibal Mukhopadhyay for reviewing my thesis and commenting on it.

I would like to thank all embedded systems lab members, Chung-Ching Lin, Honggab Kim, Ping-Chang Shih, Sehun Kim, and Tai-Ming Lin for their friendship and supports.

I also would like to thank Nagesh Lakshminarayana for our discussions regarding GPU architectures and simulators. I am thankful to Hsiang-Huang Wu of University of Maryland at College Park for his friendship and our discussions regarding graph representations of DSP applications. I am also thankful to Neal Bambha of US Army Research Laboratory for providing his graph scheduling simulator.

This dissertation is dedicated to my family. They have been extremely supportive of me throughout my life. I would like to express my deepest gratitude to my wife, Minji Kim and my daughter, Evelyn Jiahn Lee. Finally I would like to appreciate my parents for their endless supports and love. Without their supports, I would not be the person I am today.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The main objective of this thesis is to propose new methods for designing high-performance embedded computer system architectures. To achieve the goal, three major components - multi-core processing elements (PEs), DRAM main memory systems, and on/off-chip interconnection networks - in multi-processor embedded systems are examined in each section respectively.

The first section of this thesis presents architectural enhancements to graphics processing units (GPUs), one of the multi- or many-core PEs, for improving performance of embedded applications. An embedded application is first mapped onto GPUs to explore the design space, and then architectural enhancements to existing GPUs are proposed for improving throughput of the embedded application.

The second section proposes high-performance buffer mapping methods, which exploit useful features of DRAM main memory systems, in DSP multi-processor systems. The memory wall problem becomes increasingly severe in multiprocessor environments because of communication and synchronization overheads. To alleviate the memory wall problem, this section exploits bank concurrency and page mode access of DRAM main memory systems for increasing the performance of multiprocessor DSP systems.

The final section presents a network-centric Turbo decoder and network-centric FFT processors. In the era of multi-processor systems, an interconnection network is another performance bottleneck. To handle heavy communication traffic, this section applies a crossbar switch – one of the indirect networks – to the parallel Turbo decoder, and applies a mesh topology to the parallel FFT processors. When designing the mesh FFT processors, a very different approach is taken to improve performance; an optical fiber is used as a new interconnection medium.

CHAPTER I

INTRODUCTION

1.1 Motivations

Embedded applications are defined as computer applications that provide dedicated functions, sometimes with the constraints of real-time responses. They are contrary to general-purpose applications that provide a wide range of functions. DSP-oriented applications (such as jpeg and mpeg) and communication-oriented applications (such as Turbo decoding and rake receiving) are examples of embedded applications. These applications are increasingly important due to the prevalence of mobile devices such as smart phones.

As the computational complexity of embedded applications grows, a new computing paradigm other than single processor-based systems is needed to meet increasing performance requirements. Even though single-processor systems can satisfy increasing throughput requirements by deepening pipelines or by increasing an operation frequency, single-processor systems have the power wall and development/verification time problems. To overcome these problems, multi-processor systems with relatively simple component cores (i.e., in-order cores) can be an alternative. Multiprocessor systems are widely used in both general-purpose and embedded computing to keep up with Moore's law.

In the era of multi-processor systems, performance bottlenecks move from computations to communications. Therefore, efficient designs of communication components such as memory systems and interconnection networks are critical.

1.2 Overview of dissertation

We propose new methods for designing high-performance embedded computer system architectures. We deal with three main components in multi-processor embedded

systems: multi-core processing elements (PEs), DRAM main memory systems, and on/off-chip interconnection networks.

When implementing embedded applications onto hardware platforms, optimizations on non-functional metrics such as throughput, power consumption, and area are essential for the embedded systems to be practically useful. In this thesis, we focus on throughput and power consumption.

First, we propose architectural enhancements to graphics processing units (GPUs), one of the multi- or many-core processing elements, for improving performance of embedded applications. The GPU was originally developed for graphics, but is being used for general-purpose computing as well (this trend is called general-purpose computing on GPUs (GPGPU)). In this thesis, we map an embedded application onto GPUs and explore the design space of the application. We eventually propose architectural enhancements to existing GPUs for improving throughput.

Second, we propose high-performance buffer mapping methods that exploit useful features of DRAM main memory systems in DSP multi-processor systems. As a performance gap between a processor and a memory system increases, the memory system becomes a performance bottleneck. This memory wall problem becomes increasingly severe in multiprocessor environments because of communication and synchronization overheads. A hierarchical memory system including caches and useful features of the DRAM main memory system (such as bank concurrency and page mode access) can alleviate the memory wall problem; however, effective analysis and utilization of such memory systems are challenging. In this thesis, we study the latter form of memory system enhancements – especially use of DRAM bank concurrency for increasing the performance of multiprocessor DSP systems.

Finally, we propose a network-centric Turbo decoder and network-centric FFT processors. In the era of multi-processor systems, an interconnection network is another performance bottleneck. To handle heavy communication traffic, we design efficient

interconnection networks. We apply a crossbar switch – one of the indirect networks – to our parallel Turbo decoder, and we apply a mesh topology to our parallel FFT processors. When designing the mesh FFT processors, we take a very different approach to improve performance; we use an optical fiber as a new interconnection medium.

1.3 History

1.3.1 Multi- or many-core processing elements

In 1965, Intel co-founder Gordon E. Moore observed that the number of transistors on an integrated circuit doubles approximately every two years [1]. This Moore's law has been valid for almost a half century in a single-core processor paradigm.

Because of the power wall problem (finally enormous heat dissipation), a single-core-based design method runs out of steam. Multi- or many-core systems with relatively simple component cores can be a solution to overcome the limitation and keep Moore's law on track in the future.

In 2001, IBM released POWER4, the first multi-core (dual-core) processor [2]. In 2004 ~ 2005, Intel and AMD unveiled their first multi-core processors, Pentium D (dual-core, code name of Smithfield) and Athlon 64 X2 (dual-core) respectively.

Despite of different target applications, DSPs have a similar trend as general-purpose CPUs. In 1980, the first stand-alone, complete DSPs – the NEC μPD7720 and AT&T DSP1 – were made public. Since then, Texas Instruments released the three-core TMS320C6488 and four-core TMS320C5441, and Freescale released the four-core MSC8144 and six-core MSC8156.

The DSP has been widely used as processing elements when mapping embedded applications onto programmable processors to achieve high performance with flexibility (programmability). DSP solutions generally provide lower performance than ASIC solutions.

For example, some previous work implemented the Turbo decoding algorithm, one of the computationally intensive communication applications, in industrial DSP cores such as TI C6x and Starcore SC140 [3, 4]. Unfortunately, their throughput is below the standards' requirements. For example, none of them satisfy the 2 Mbps requirement of the WCDMA standard.

To support both high performance and flexibility in embedded computing applications, we consider another type of programmable core – GPUs − in this thesis. Especially, GPGPUs are of our interest. GPUs have evolved from fixed pipeline graphics hardware to programmable and unified graphics hardware. GPUs are designed for highly parallel tasks and assign more transistors for computations than for controls. As a result, GPUs provide many number of processing elements that are relatively simple in-order cores compared to CPUs.

In this thesis, we propose architectural enhancements to GPUs specialized for embedded applications; i.e., we propose GPGPU architectures.

1.3.2 Main memory systems

Memory systems are hierarchically constructed in high-performance computing systems to overcome the memory wall problem. Main memory systems, which are composed of memory itself and memory controllers, are generally placed at a level between single- or multi-level caches and hard disks.

For several decades, DRAMs have been widely used as a main memory. In 1966, Dennard at the IBM Thomas J. Watson Research Center invented a DRAM awarded U.S. patent number 3,387,286 in 1968. In contemporary DRAMs, a single transistor-capacitor pair constitutes single bit storage. The "dynamic" term originated from the fact that the capacitor requires periodic refreshes for data retention.

As a processor speed increases and memory latency-tolerating techniques emerge, conventional DRAM architectures need to be improved for providing a higher memory bandwidth. Fast page mode DRAM (FPM DRAM) [5], extended data-out DRAM (EDO DRAM) [6], synchronous DRAM (SDRAM) [7] − SDRx and DDRx SDRAMs –, Rambus DRAM (RDRAM) [8], and Direct Rambus DRAM (DRDRAM) [9] are examples of advanced DRAM architectures.

Due to non-trivial capacitor and transistor size requirements, a DRAM is known to be non-scalable in terms of its capacity and density beyond a 40 nm process technology [10]. Instead, a phase change memory (PCM) provides a non-volatile, scalable storage mechanism. A PCM was demonstrated in a 20 nm prototype and is expected to scale down to 9 nm [10, 11]. However, its large access latency should be overcome to replace a DRAM as a main memory component. A PCM is out of our scope.

A memory controller is another critical component of main memory systems. In general-purpose computing, most of the research on improving memory performance has focused on memory controller techniques, such as command scheduling and memory address interleaving [12, 13, 14]. Several techniques [15, 16], which utilize both bank concurrency and page mode, exist to reorder memory commands in a memory controller.

In addition to hardware-oriented approaches − i.e., designing high-performance memory cores and memory controllers −, software-oriented approaches are another type of solution to mitigate the memory wall problem.

In this thesis, we propose a high-level compiler (converting graphs to C codes) technique to exploit useful features of DRAM main memory systems.

1.3.3 Interconnection networks

With the advent of multi- or many-core processing elements, an interconnection network is another performance bottleneck. In a micro-architecture community, on-chip

interconnects are major concerns – e.g., interconnection network designs in the network-on-a-chip (NoC) of a chip multiprocessor (CMP). In this thesis, we consider both on-chip and off-chip interconnection networks. By off-chip interconnection networks, we mean an interface between cores and off-chip main memory systems, unless otherwise stated (we assume that memory controllers are off-chip).

Although constraints – such as bandwidth, latency, timing, and area – are different between on-chip and off-chip networks, techniques that have been used in one network can be applied to another. Various topologies − such as mesh, crossbar, trees, and Benes network [17] − have been proposed for computer systems. Over the past 20 years, *k*-ary *n*-cube (torus) [18] topology has been extensively used in systems such as Intel/CMU iWarp [19], Cray T3D [20], and Cray T3E [21].

However, electrical networks have disadvantages of low bandwidth density [in bps/mm], distance-dependent power consumption, and electro-magnetic interference. An optic interconnect is a good replacing candidate − it has high bandwidth density, distance-independent power consumption, and no interference.

In 1984, Goodman explored the feasibility of optic usage as a communication medium in IC and system designs [22]. Main barriers against optic usage were CMOS incompatibility and expensive electrical-optical translation costs. Owing to the rapid progress in a nano-photonic technology, these problems are now resolved.

Optical topologies and routing algorithms have been recently studied in NoC designs [23, 24, 25]. Bergman et al. demonstrated the functional correctness of off-chip optical interconnects experimentally by setting up real test-beds [26, 27]. In their experiments, microprocessors write a bunch of data to SDRAM main memory systems through optical networks, read them back, and compare them with the original source data. They also demonstrated 70% improvements in static power consumption compared to that of off-chip electrical interconnects.

In this thesis, we propose a network-centric parallel Turbo decoder and network-centric parallel FFT processors. The FFT processors use both electrical and optical media as an interconnection medium.

CHAPTER II

MULTI-CORE PROCESSOR – GPUS

Our goal in this section is to propose architecture enhancements to the existing GPUs for improving performance of DSP applications. First, we introduce GPU fundamentals and explore the design space of one example application on GPUs to find performance bottlenecks. Finally, we propose architectural enhancements to existing GPUs for improving performance.

2.1 GPU fundamentals

Instruction-level and data-level parallelisms are generally exploited in embedded applications. Since some embedded applications, such as DSP applications, generally have a static control flow – i.e., a control flow can be determined during a compile-time – , a compiler can schedule instructions so that a set of instructions can be issued simultaneously (instruction-level parallelism). In addition, an instruction or a set of instructions can be issued to several sets of data (data-level parallelism). Therefore, most DSP processors fall into one of the two types of machine: very long instruction word (VLIW) and/or single instruction multiple data (SIMD). VLIW machines support the static instruction-level parallelism and SIMD machines support the data-level parallelism. GPUs extensively support the data-level parallelism across multiple cores.

A high-performance programmable GPU, such as the Tesla architecture GPU [28] is a good candidate for our work. It was originally developed for graphics, but is widely used in general-purpose computing today (this trend is called GPGPU). In addition to its huge number of processing elements, the GPU provides a good programming model as well. Our work is based on the assumption that mobile GPUs would be available in

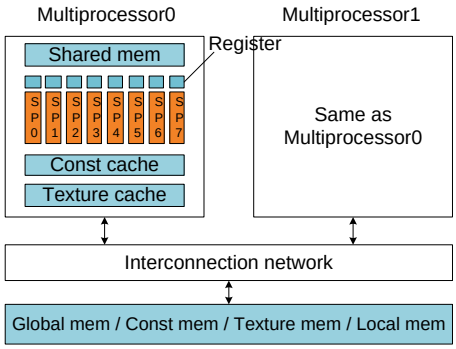


Figure 1. Hardware architecture of the ION GPU.

mobile embedded systems such as smart phones in the near future. We consider the ION GPU, the simplest chip among NVIDIA GPUs, as our baseline architecture.

2.1.1 Hardware

GPUs have evolved from fixed pipeline graphics hardware to programmable and unified graphics hardware. GPUs are designed for highly parallel tasks and assign more transistors for computations than for controls. As a result, GPUs provide many number of processing elements that are relatively simple in-order cores compared to CPUs. We consider NVIDIA GPUs in this thesis. All NVIDIA GPUs have fundamentally the same hardware architecture.

As shown in Fig. 1, the ION GPU is composed of two multi-threaded streaming multiprocessors (SMs), and each multiprocessor is composed of eight homogeneous scalar processors (SPs). The ION GPU supports different kinds of memories in terms of scope, latency, and size. We should note that the local memory in Fig. 1 is physically located off-chip despite of its scope to per thread. Therefore, to avoid performance degradation due to memory access latency, application programmers should avoid using the local memory and global memory as much as possible. The number of registers per multiprocessor is 8192, the amount of shared memory per multiprocessor is 16 kB, and the total amount of constant memory is 64 kB [29].

2.1.2 Programming model

The compute unified device architecture **(**CUDA) programming model is a refinement of the vector programming model to enhance arithmetic intensity. In the CUDA programming model, task executions are done by a hierarchical structure of threads – the largest granularity is called a *kernel* that is composed of several thread blocks, and each *thread block* is composed of several threads. A GPU *thread* is more light-weight than a CPU thread, so one can generate enough number of threads to hide memory access latency and other kinds of latency. The granularity of task assignments onto multiprocessors is a thread block and the granularity of instruction scheduling and execution is a *warp* referring to a set of 32 threads. The CUDA programming model helps application programmers focusing on the parallelism analysis of applications rather than programming – application programmers only need to write a kernel code for one thread, which is automatically executed by all threads in a kernel.

2.1.3 Parameters to affect performance

In this sub-section, we introduce several major factors to determine application performance on GPUs. We will use these definitions throughout this thesis.

In GPUs, the *occupancy* is defined as the ratio of the number of active warps per multiprocessor to the maximum number of active warps per multiprocessor, where an active warp is a warp that can be handled by a multiprocessor at a time. According to its definition, the occupancy indicates how fully given multiprocessor resources are utilized. If the current warp is stalled because of memory access latency and arithmetic operation latency, a multiprocessor can hide the latency by executing another warp to be ready – i.e., the warp whose operands are available. Therefore, the higher occupancy, the larger pool of candidate warps to be picked.

Three factors interactively determine the occupancy: the number of threads per thread block, a shared memory requirement per thread block, and a register requirement

per thread. We should note that those factors interact – e.g., increasing the number of threads (positive on the occupancy) can increase shared memory and register requirements (negative on the occupancy). In the worst case, if a multiprocessor does not have enough shared memory and register resources for one block to be launched at least, a kernel fails to be launched. Therefore, we should carefully determine these three parameters to get a maximum occupancy.

In the ION GPU, the shared memory is configured in a 16 multi-bank structure enabling parallel memory access. If threads within a half warp try to access the same memory bank simultaneously, bank conflicts occur. In this case, memory requests are divided into several conflict-free requests that are served sequentially.

Global memory access by all threads of a half-warp can be coalesced if access patterns satisfy some requirements [29]. If not, global memory requests are separated into several memory transactions, which results in reduced performance.

If threads within a warp follow different execution paths, the paths are executed sequentially (called a warp divergence), which also has a negative effect on performance.

2.2 Design space exploration of embedded applications mapped onto GPUs

To propose architectural enhancements to GPUs, we first find performance bottlenecks in embedded applications mapped onto GPUs.

In this section, we select Turbo decoding as a target application since it is one of the very computationally intensive components in wireless communication systems. Examples of Turbo code applications are WCDMA, NASA missions such as Mars Reconnaissance Orbiter, IEEE 802.16 (WiMAX), and Qualcomm's MediaFLO [30]. Among the applications, we choose the WCDMA standard and try to achieve its 2 Mbps requirement on the GPU.

We consider three axes for the design space exploration: a radix degree, a parallelization method, and the number of sub-frames per thread block. In Turbo

decoding, a degree of radix affects computational complexity and memory access patterns in both algorithmic and implementation viewpoints. Second, computations of branch metrics (BrMs) and state metrics (StMs) have a different degree of parallelism, which affects a mapping method of computational tasks to GPU threads. Finally, we can easily adjust the number of sub-frames per thread block to balance the occupancy and memory access traffic.

2.2.1 Turbo decoding algorithm

2.2.1.1 Log-MAP algorithm

The Maximum *A Posteriori* (MAP) algorithm was proposed by Bhal et al. in 1974 [31]. It provides optimal BER performance at the cost of high computational complexity. Final output metrics of the MAP algorithm are Log-Likelihood Ratios (LLRs), the logarithm of the ratio of *a posteriori* probability (APP) of information bit u_k being '1' to that of u_k being '0':

$$\Lambda(u_k) = \ln \frac{P(u_k = 1 \mid y)}{P(u_k = 0 \mid y)}. \tag{1}$$

At the last stage of a Turbo decoder, a decision unit determines a transmitted information bit as '0' or '1' according to the corresponding LLR's sign – if LLR(u_k) is positive, the transmitted information bit, u_k, is decoded as '1'; otherwise it is decoded as '0'.

For practical computations of LLRs, Eq. (1) can be rewritten by using three metrics as

$$\Lambda(u_k) = \ln \frac{\sum_{(S_{k-1}, S_k): u_k = 1} \alpha_{k-1}(S_{k-1}) \gamma_k(S_{k-1}, S_k) \beta_k(S_k)}{\sum_{(S_{k-1}, S_k): u_k = 0} \alpha_{k-1}(S_{k-1}) \gamma_k(S_{k-1}, S_k) \beta_k(S_k)}, \tag{2}$$

where S_{k-1} and S_k are the trellis states of a component encoder at $t = k-1$ and $t = k$ respectively. Gamma called a branch metric is calculated as follows:

$$\gamma_k(S_{k-1}, S_k) = y_k \times u_k + y_k^p \times u_k^p + \ln \Pr\{S_k \mid S_{k-1}\}, \tag{3}$$

where u and u^p are information and parity bits respectively. y and y^p are the corresponding received symbols. Alpha is a forward-recursion state metric, and beta is a backward-recursion state metric expressed as follows:

$$\alpha_k(S_k) = \sum_{S_{k-1}} \alpha_{k-1}(S_{k-1})\,\gamma_k(S_{k-1}, S_k), \tag{4}$$

$$\beta_k(S_k) = \sum_{S_{k+1}} \beta_{k+1}(S_{k+1})\,\gamma_k(S_k, S_{k+1}). \tag{5}$$

The Log-MAP algorithm can reduce computational complexity of the MAP algorithm by calculating logarithms of alpha, beta, and gamma as follows [32]:

$$\begin{aligned} \bar{\alpha}_k(S_k) = \ln \alpha_k(S_k) &= \ln\Big(\sum_{S_{k-1}} e^{\ln \alpha_{k-1}(S_{k-1}) + \ln \gamma_k(S_{k-1}, S_k)}\Big) \\ &= \max_{S_{K-1}}{}^{*}\{\bar{\alpha}_{k-1}(S_{k-1}) + \bar{\gamma}_k(S_{k-1}, S_k)\}, \end{aligned} \tag{6}$$

$$\begin{aligned} \bar{\beta}_k(S_k) = \ln \beta_k(S_k) &= \ln\Big(\sum_{S_{k+1}} e^{\ln \beta_{k+1}(S_{k+1}) + \ln \gamma_k(S_k, S_{k+1})}\Big) \\ &= \max_{S_{k+1}}{}^{*}\{\bar{\beta}_{k+1}(S_{k+1}) + \bar{\gamma}_k(S_k, S_{k+1})\}. \end{aligned} \tag{7}$$

Then by substituting Eqs. (6) and (7) into Eq. (2), an LLR is computed as

$$\begin{aligned} \Lambda(u_k) = &\max_{(S_{k-1}, S_k): u_k = 1}{}^{*}\{\bar{\alpha}_{k-1}(S_{k-1}) + \bar{\gamma}_k(S_{k-1}, S_k) + \bar{\beta}_k(S_k)\} \\ &- \max_{(S_{k-1}, S_k): u_k = 0}{}^{*}\{\bar{\alpha}_{k-1}(S_{k-1}) + \bar{\gamma}_k(S_{k-1}, S_k) + \bar{\beta}_k(S_k)\}, \end{aligned} \tag{8}$$

where $\max_{i \in \{1,\ldots,n\}}{}^{*} \delta_i = \ln\big(\sum_{i=1}^{n} e^{\delta_i}\big)$ (called log-sum). The multiplications and divisions in Eqs. (2), (4), and (5) of the MAP algorithm are now replaced with the additions and subtractions in Eqs. (6), (7), and (8) of the Log-MAP algorithm. Although the log-sum introduces another computational complexity, one can simply compute the log-sum by applying the Jacobian logarithm, $\ln(e^{\delta_1} + e^{\delta_2}) = \max\{\delta_1, \delta_2\} + \ln(1 + e^{-|\delta_1 - \delta_2|})$ [33]; recursions are applied if the number of arguments in the log-sum is greater than two. The second term, ln(-), is called a correction term that is generally obtained by a lookup table.

Since the Log-MAP algorithm provides reduced computational complexity with BER performance close to that of the MAP algorithm within 0.05 dB [34], we consider the Log-MAP algorithm in this section. Henceforth, a plain (i.e. without the upper bar)

alpha, beta, and gamma represent a log version of alpha, beta, and gamma for simple descriptions.

2.2.1.2 Radix-4 algorithm

We consider a radix degree – radix-2 or radix-4 - as the first axis for the design space exploration.

In binary Turbo codes, the normal Log-MAP algorithm is based on radix-2. By applying a two-level look-ahead transformation, the radix-2 algorithm is transformed to the radix-4 algorithm as follows (for the detailed derivation, we refer readers to [32]):

$$\alpha_k(S_k) = \max_{S_{k-2}}^{*}\{\alpha_{k-2}(S_{k-2}) + \gamma_k(S_{k-2}, S_k)\}, \tag{9}$$

$$\beta_k(S_k) = \max_{S_{k+2}}^{*}\{\beta_{k+2}(S_{k+2}) + \gamma_k(S_k, S_{k+2})\}, \tag{10}$$

$$\begin{aligned}\Lambda(u_{k-1}) &= \max^{*}\{\Gamma_{10}, \Gamma_{11}\} - \max^{*}\{\Gamma_{00}, \Gamma_{01}\},\\ \Lambda(u_k) &= \max^{*}\{\Gamma_{01}, \Gamma_{11}\} - \max^{*}\{\Gamma_{00}, \Gamma_{10}\},\end{aligned} \tag{11}$$

where Γ_{ij} is a 2-bit symbol reliability metric with the first information bit $u_k = i$ and the second information bit $u_{k-1} = j$, which is represented as

$$\Gamma_{ij} = \max_{(S_{k-2}, S_k)}^{*}\{\alpha_{k-2}(k-2) + \gamma_k^{ij}(S_{k-2}, S_k) + \beta_k(S_k)\}, \tag{12}$$

where γ_k^{ij} is the branch metric of the radix-4 branch with $u_k = i$ and $u_{k-1} = j$.

The radix-4 algorithm can be explained more clearly by comparing its trellis diagram with a radix-2 diagram. Figure 2 depicts the radix-2 and radix-4 trellis diagrams of the 3rd Generation Partnership Project (3GPP) Turbo encoder [35] as an example. As shown in the figure, two trellis stages in the radix-2 algorithm are grouped into one trellis stage in the radix-4 algorithm. Therefore, in the radix-4 algorithm, computations of alpha and beta metrics at $t = k$ require alpha and beta metrics at $t = k$-2 and $t = k$+2, respectively, as shown in Eqs. (9) and (10). The radix-4 algorithm decodes two information bits (u_{k-1} and

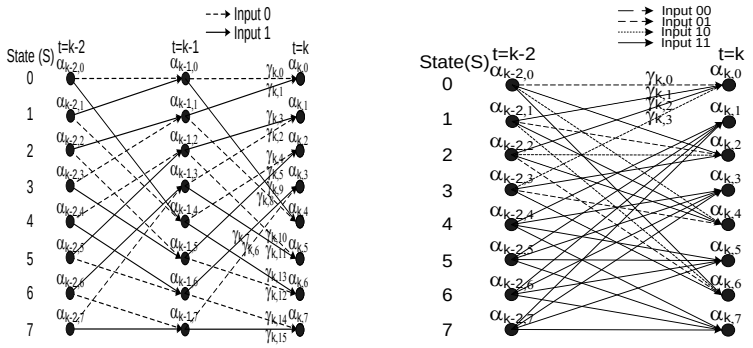


Figure 2. Radix-2 and radix-4 trellis diagrams.

u_k) per radix-4 decoding cycle, as shown in Eq. (11). We consider "radix-2 or radix-4" as the first axis for the design space exploration.

2.2.2 Parallelism analysis

To exploit parallelisms available in the Turbo decoding algorithm, we need to identify data dependencies. The concept of iterations in Turbo decoding is central to explaining data dependencies. One set of computations corresponding to one forward and backward traversal through the whole trellis diagram constitutes one half-iteration. Another set of computations corresponding to another forward and backward traversal constitutes another half-iteration.

Data dependencies exist across half-iterations and within each half-iteration. Both inter-half-iteration and intra-half-iteration data dependencies are strong in the Turbo decoding algorithm − i.e., computations of the current half-iteration require computation results of the previous half-iteration, and computations at the current trellis stage require computation results at the previous trellis stage within a half-iteration. In this section, a trellis stage and a trellis state are defined as follows; each time stamp indicates one trellis stage that is composed of several trellis states. For example, Fig. 3 depicts six trellis stages and each stage is composed of four trellis states. Since Turbo decoding has both types of data dependencies, it is very difficult to parallelize the decoding sequence and

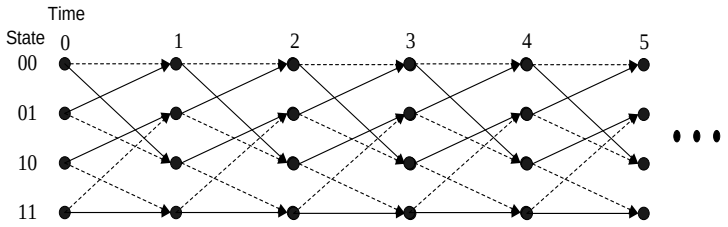


Figure 3. Trellis diagram representation of Turbo decoding.

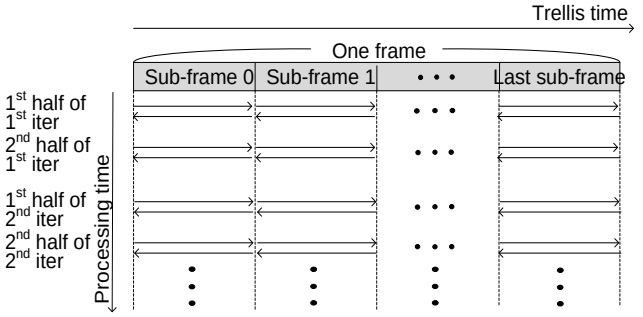


Figure 4. Turbo decoding sequence.

map the Turbo decoding algorithm onto GPUs. Both data dependencies are inevitable in the sense that if we remove these dependencies to attain high throughput, it leads to a huge BER performance loss.

Fortunately, other kinds of parallelisms can be exploited in the Turbo decoding algorithm. At the most coarse-grained level, one can exploit a frame-level parallelism, where several frames are processed simultaneously by several Turbo decoders. At the next level, one can use a sub-frame-level parallelism, where a frame is divided into several sub-frames, as shown in Fig. 4, and all sub-frames are processed in parallel, each by its dedicated MAP core. At the most fine-grained level, a trellis state-level parallelism can be exploited. In Fig. 2, computations corresponding to all states at each trellis stage are independent of each other and can be processed concurrently. The trellis state-level parallelism fits very well to the SIMD style of the GPU since operations of all states or all branches are exactly same, but to different sets of input data. Since the frame-level parallelism shows the largest decoding latency among the three, we consider the sub-frame-level and trellis state-level parallelisms in this section.

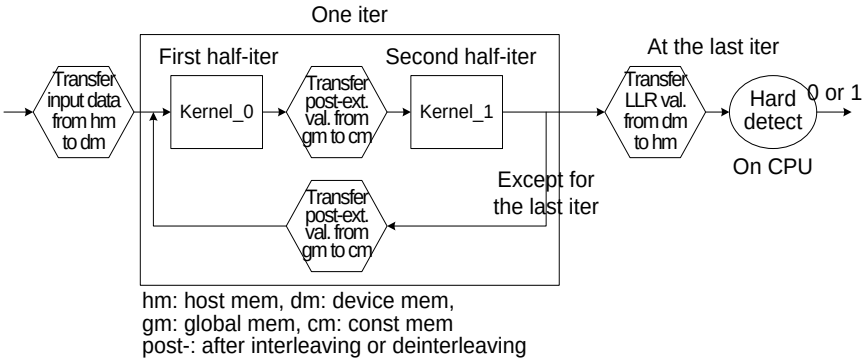


Figure 5. Overall mapping.

2.2.3 Overall mapping

Figure 5 shows an overall mapping of the Turbo decoding algorithm onto the GPU in our implementation. In Turbo decoding, two half-iterations of a single iteration communicate extrinsic values and need a complete synchronization of all threads. Therefore, computations of the first half-iteration are assigned to one kernel (kernel_0) and computations of the second half-iteration are assigned to another kernel (kernel_1).

The CPU (host) transfers raw input data from the host memory to the device memory, and then it calls kernel_0. The interleaving of extrinsic values is done within kernel_0 by writing the values to the global memory in a permuted order. After the GPU (device) completes kernel_0, it transfers interleaved extrinsic values from the global memory to the constant memory to take advantage of the constant cache. Then the device executes kernel_1. The deinterleaving is done within kernel_1 in the same way as the interleaving in kernel_0. These steps are repeated as many as the number of iterations. At the last iteration, the GPU generates LLR values and transfers them from the device memory to the host memory, and then the CPU executes hard-detections, completing a whole process for one frame.

Since the interleaving/deinterleaving (I/DI) does not provide an opportunity of increasing a parallelism, it can be executed on either the CPU or the GPU. The I/DI on

the CPU requires data transfers through the PCI bus between the device memory and the host memory. On the other hand, the I/DI on the GPU requires data transfers through the internal data bus between the GPU and the device memory. Since, in our system, a bandwidth of the internal bus (25.6 GB/sec) is much larger than that of the PCI bus (PCI-Express x16 ver.2: 8 GB/sec), we run the I/DI on the GPU.

2.2.4 Memory allocation

We allocate main variables of Turbo decoding to specific memory types according to the variables' scope and read/write attribute, as shown in Fig. 6.

Read/write attributes in this paragraph are from the viewpoint of the GPU (device). Our allocation strategy is that the read-only data transferred from the host to the device (such as raw input data and interleaved extrinsic values) is allocated to the constant memory. The read/write or write-only data transferred from/to the host to/from the device (such as extrinsic values and at the last iteration, LLR values) is allocated to the global memory. The data shared by threads of a single thread block (such as alpha, beta, and gamma metrics) is allocated to the shared memory. The alpha, beta, and gamma metrics are shared by threads of a thread block according to data permutation patterns between trellis stages. We do not use the texture memory.

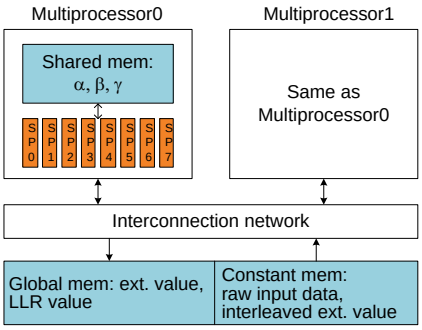


Figure 6. Memory allocations.

2.2.5 Task assignments considering computational parallelisms

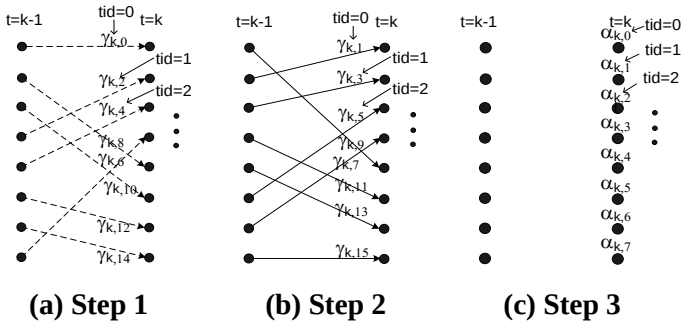
We exploit two kinds of computational parallelisms – trellis state-level and sub-frame-level parallelisms. In this sub-section, we define additional two axes – one related to the trellis state-level parallelism and the other related to the sub-frame-level parallelism – for the design space exploration.

2.2.5.1 Trellis state-level parallelism

Although the Turbo decoding algorithm has very strong data dependency between trellis stages (intra-half-iteration dependency), all StMs or all BrMs at each trellis stage can be computed concurrently. For example, at a trellis stage $t = k$ in the radix-2 trellis diagram of Fig. 2, all 16 BrMs ($\gamma_{k,0}$ to $\gamma_{k,15}$) or all 16 (previous StM + BrM)s ($\alpha_{k-1,0}+\gamma_{k,0}$ to $\alpha_{k-1,15}+\gamma_{k,15}$ – these results are arguments of the max* operation in Eq. (6)) can be computed in parallel and then all eight StMs ($\alpha_{k,0}$ to $\alpha_{k,7}$) can be computed in parallel as well. Henceforth, we refer to both BrM and (previous StM + BrM) computations as a BrM computation and refer to only a max* operation as a StM computation.

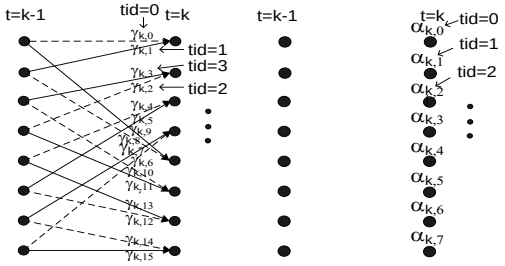
Since the parallelization degree of BrM and StM computations is different – the degree in BrM computations is radix times that in StM computations –, how to map computations to GPU threads is another important parameter that affects performance. This different degree of parallelism is always an issue in applications where a core algorithm can be represented as a trellis diagram. We consider a parallelization method (i.e., a mapping method) as the second axis for the design space exploration (the first axis is a degree of radix). We call two candidate mapping methods reflecting the different parallelization method StM-centric and BrM-centric mappings.

In the StM-centric mapping, the number of threads per thread block is equal to the number of states per trellis stage, which is less than the number of branches. As a result, a part of BrMs are computed sequentially while all StMs are computed at a time. Figure 7 depicts an example of thread assignments to metric computations in the StM-centric



(a) Step 1 **(b) Step 2** **(c) Step 3**

Figure 7. Thread assignments in StM-centric mapping of the radix-2 algorithm.



(a) Step 1 **(b) Step 2**

Figure 8. Thread assignments in BrM-centric mapping of the radix-2 algorithm.

method of the radix-2 algorithm. In this example, the number of threads per block is eight, so 16 BrMs are computed in two separate steps. Then, all eight StMs are computed at a time. More specifically, at $t = k$, thread_0 sequentially computes the two BrMs ($\gamma_{k,0}$ and $\gamma_{k,1}$) incoming to state_0 and then computes $\alpha_{k,0}$. Thread_1 sequentially computes the two BrMs ($\gamma_{k,2}$ and $\gamma_{k,3}$) incoming to state_1 and then computes $\alpha_{k,1,}$ and so on. By assigning a pair of two branches with the same destination state to the same thread, we can avoid communication and synchronization overheads of BrM computation results across threads. In other words, BrMs are stored to and loaded from the local registers, not the shared memory.

In the BrM-centric mapping, we fully parallelize BrM computations, i.e., the number of threads per thread block is equal to the number of branches per trellis stage. All threads do work during BrM computations, but some of them are in an idle state during StM computations. Figure 8 shows an example of thread assignments in the BrM-centric

method of the radix-2 algorithm. In this example, the number of threads per thread block is 16. All BrMs are computed at a time by 16 threads, and then all StMs are computed at a time by eight threads. The other eight threads are not used during the StM computations. More specifically, at $t = k$, thread_0 computes $\gamma_{k,0}$ and thread_1 computes $\gamma_{k,1}$. After all threads complete BrM computations, thread_0 computes $\alpha_{k,0}$ and thread_1 computes $\alpha_{k,1}$. There are two different points compared to the StM-centric mapping method. BrMs must be shared among threads, so they are stored to and loaded from the shared memory. StM computations can be started only after all threads complete all BrM computations, so a synchronization barrier is required.

The two mapping methods in the radix-4 algorithm have similar characteristics as them in the radix-2 algorithm. For example, in the StM-centric mapping of the radix-4, the number of required steps for BrM computations is four – a quarter of BrMs are computed at every step, and each thread computes four BrMs sequentially. On the other hand, in the BrM-mapping method of the radix-4, all BrMs are computed at step 1 at a time.

In the following paragraphs, we compare the two mapping methods in terms of the occupancy, warp divergence, and memory access patterns.

First, it is hard to decide which mapping method provides a higher occupancy. Three factors determine the occupancy: the number of threads per thread block, a shared memory requirement, and a register requirement. The two mapping methods require almost the same shared memory size. As shown in Fig. 6, we allocate the shared memory to alpha, beta, and gamma metrics. Among the three, alpha metrics occupy most of the shared memory space (this will be explained in Section 2.2.6). The alpha memory requirement is

(*# of states per trellis stage*) x (*sub-frame length*) x (*# of sub-frames per thread block*) x (*word length of data*). (13)

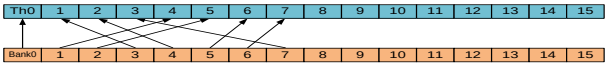
According to Eq. (13), the mapping method itself does not affect the alpha memory requirement. In terms of the number of threads per thread block (the second factor to determine the occupancy), the BrM-centric method is better than the StM-centric method. But increasing number of threads might increase the register requirement (the third factor to determine the occupancy). Therefore, we cannot easily decide which mapping method provides a higher occupancy. In Section 2.3.1, we will compare the occupancy of the two mapping methods based on experimental results.

Second, both mapping methods have no warp divergence. In the StM-centric mapping, this is achieved at the cost of serialization of BrM computations. The BrM-centric mapping also has no warp divergence; Although some threads of a warp are in an idle state during StM computations, these idle threads do not contribute to the warp divergence (they just do nothing).

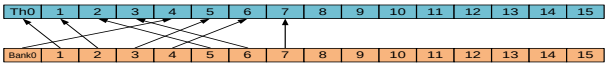
Before describing three factors related to memory access – shared memory bank conflicts, global memory coalescing, and constant memory caching –, we should note that all three factors are intra-half-warp parameters. In other words, memory access patterns of 16 threads of a half-warp are only meaningful, but the patterns across different half-warps are meaningless.

Third, the BrM-centric method shows shared memory bank conflicts, but the StM-centric method does not. Figures 9 and 10 show shared memory access patterns for the example of Figs. 7 and 8, respectively. (In our implementation, the data type of all three metrics is a 32-bit floating point.) In the StM-centric method, although there are data permutations during steps 1 and 2, bank conflicts do not occur. On the other hand, in the BrM-centric method, two-way bank conflicts occur during α_{k-1} loads at step 1.

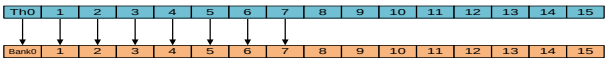
Fourth, the two mapping methods have the equal number of global memory accesses and the same coalescing factor. Our implementation uses the global memory only for storing extrinsic information (or LLR values at the last iteration). Since only one extrinsic



(a) Step 1: α_{k-1} load

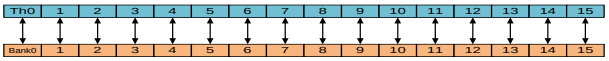


(b) Step 2: α_{k-1} load

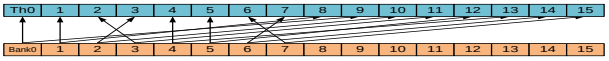


(c) Step 3: α_k store

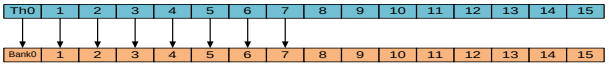
Figure 9. Shared memory access patterns in StM-centric mapping of Fig. 7 (Gamma metrics are loaded/stored from/to registers).



(a) Step 1: γ_k store and load



(b) Step 1: α_{k-1} load (two-way conflict)



(c) Step 2: α_k store

Figure 10. Shared memory access patterns in BrM-centric mapping of Fig. 8.

value (or one LLR value) is generated and stored to the global memory at each trellis stage, the total number of global memory accesses is calculated regardless of the mapping method as follows:

(*Total # of gmem inst*) = (*frame length*) x (*2 half-iterations*) x (*# of iterations*) / (*coalescing factor*). (14)

In addition, the stride of global memory accesses from two adjacent threads within a half-warp is determined by sub-frame length, e.g., if the first thread gets access to address 128, the second thread gets access to address (128 + 4 x sub-frame length). (In our implementation, the data type of an extrinsic information and LLR value is a 32-bit floating point.) This amount of stride causes non-coalesced memory accesses and the

coalescing factor, which is defined as the number of global memory instructions processed in one unit, is not affected by the mapping method.

Fifth, the two mapping methods have the equal number of constant memory accesses and the same degree of data access locality. Our implementation mainly uses the constant memory for raw input data and interleaved extrinsic values. As in the global memory access, the number of constant memory accesses is affected by frame length, not by the mapping method. In addition, each frame is iteratively processed (due to an iterative decoding feature of the Turbo decoding algorithm) and memory access patterns are sequential, so temporal and spatial localities are very good. We will show a hit ratio of constant cache accesses in Section 2.3.1.

In summary, the BrM-centric method provides more threads per thread block, but it does not guarantee a higher occupancy. On the other hand, the StM-centric method shows fewer shared memory bank conflicts. The other factors – warp divergence, global memory access, and constant memory access – are same in both mapping methods.

A pseudo-code for one complete half-iteration is shown in Fig. 11, including both computations and memory transactions. The pseudo code explicitly shows the intra-half-iteration dependency and implicitly shows the trellis state-level parallelism. First, the two "*for*" loops in the forward and backward recursion sections mean strong data dependency between trellis stages. Each loop index (*i*) represents one trellis stage and SUBFRAME_LENGTH is the total number of trellis stages. The inter-loop dependency (indicated by *i*-1 in alpha computations and *i*+1 in beta computations) indicates that alpha and beta computations at the current stage can be started only after computations at the previous stage are completed. All gamma and all alpha (or all beta) metrics at each trellis stage, however, can be computed in parallel by several threads. At each trellis stage (at each loop index *i*), every thread loads input data to its own registers and then computes one or more gamma metrics (depending on the computation mapping method to threads) and one alpha metric during a forward recursion. During a backward recursion, the same

```
__global__ void turbo_kernel_0(DATA_TYPE* d_w){
alpha, beta initialization;
//Forward recursion section
for (i=1 ; i< SUBFRAME_LENGTH+1 ; i++){
    Load the raw input data from constant mem to register.
    Compute gamma metrics and store them to register or shared mem.
    __syncthreads( );
    Load the gamma metrics from register or shared mem and the previous alpha metric from
     shared mem.
    Compute a new alpha metric as alpha[i] = f(alpha[i-1], gamma[i]) and store it to shared
   mem.
    __syncthreads( );
}
//Backward recursion section
for(i= SUBFRAME_LENGTH -1 ; i>=0 ; i--){
    Load the raw input data from constant mem to register.
    Compute gamma metrics and store them to register or shared mem.
   __syncthreads( );
    Load the gamma metrics from register or shared mem and the previous beta metrics from
     shared mem.
    Compute a new beta metric as beta[i] = f(beta[i+1], gamma[i]) and store it to shared mem.
    __syncthreads( );
    Load the gamma metrics from register or shared mem and alpha, beta metrics from
     shared mem.
    Compute  a new extrinsic value (d_w) and store it to global mem in a permuted order.
   //The interleaving is done here.
   __syncthreads( );
}
```

Figure 11. Pseudo device code of one kernel.

steps are processed except that a computation direction is reversed. Finally, because of permutation requirements between trellis stages, we should insert synchronization barriers in proper locations. The locations are different in the two mapping methods because of different synchronization requirements.

The code in Fig. 11 is an example of several possible implementations.

2.2.5.2 Sub-frame-level parallelism

Although in the previous sub-section we confined our explanations to cases with only one sub-frame per thread block, multiple sub-frames can be assigned per thread block. We

call it a sub-frame-level parallelism. In some cases, assigning multiple sub-frames to a thread block is essential to increase the occupancy. For example, in Figs. 7 and 8, a single sub-frame per thread block results in the number of threads less than 32 – eight in the StM-centric mapping and 16 in the BrM-centric mapping.

The sub-frame-level parallelism fits very well to the many-core feature of the GPU. To exploit this parallelism, one frame is divided into several sub-frames and the constant number of sub-frames is assigned to each thread block. We consider the number of sub-frames per thread block as the third axis for the design space exploration.

The following paragraph analyzes the occupancy, warp divergence, and memory access patterns affected by combination of the three axes - a radix degree, a mapping method, and the number of sub-frames per thread block.

First, in all four cases, (radix-2 or 4) x (BrM- or StM-centric), the increasing number of sub-frames increases the number of threads per thread block, but also increases shared memory and register requirements. We compare the occupancy based on experimental results in Section 2.3. Second, the number of sub-frames does not affect the degree of warp divergence. In other words, the degree is only determined by (radix-2 or 4) x (BrM- or StM-centric). Finally, all three factors related to memory access – shared memory bank conflicts, global memory coalescing, and constant memory caching – are intra-half-warp parameters. Therefore, increasing the number of sub-frames does not affect those three factors at all.

These analyses are also applied to the radix-4 algorithm in the same way.

2.2.6 Shared memory usage optimizations

Another important point when mapping the Turbo decoding algorithm onto the GPU is a memory size requirement. Among the three kinds of memories in Fig. 6, we focus on handling the shared memory requirement since the shared memory is a limited resource and its size requirement is one of the key factors to determine the occupancy. The global

memory is very spacious and the constant memory requirement is directly proportional to the frame length in our implementation. Because the frame length is given by standard specifications, there is not much freedom for us to handle the constant memory requirement.

We consider three approaches to reduce the shared memory requirement: a trellis compaction, an optimization in Log-MAP algorithm implementations, and a reduction in the sub-frame length. First, the effect of the trellis compaction (transformation from the radix-2 to the radix-4) on the shared memory requirement was already explained in the previous section. Second, we can reduce the shared memory requirement by changing Log-MAP algorithm implementations. In most previous implementations, gamma and alpha metrics are computed during forward recursions. Both metrics of a whole frame are stored in a memory and are used for computing LLR values during backward recursions. (Note that the beta storage requirement can be kept very small by computing LLR values at the same time as beta computations.) A shared memory requirement in the previous implementations is

(*# of states per trellis stage*) x (*sub-frame length*) x (*# of sub-frames per thread block*) x (1 + *radix*) x (*word length of data*), (15)

where 1 of (1+radix) is for alpha metrics and the radix factor is for gamma metrics (since every state has the radix number of incoming branches, the storage requirement of gamma metrics is radix times that of alpha metrics). We assume that the word length of alpha and gamma metrics is equal. The storage for beta metrics and some temporary variables is ignored. On the other hand, in our implementation we compute gamma metrics once again during backward recursions instead of using the previously calculated and stored ones during forward recursions. By doing this, we can significantly reduce the shared memory requirement (the factor of 1+radix in Eq. (15) reduces to 1) at the cost of a small additional computation overhead.

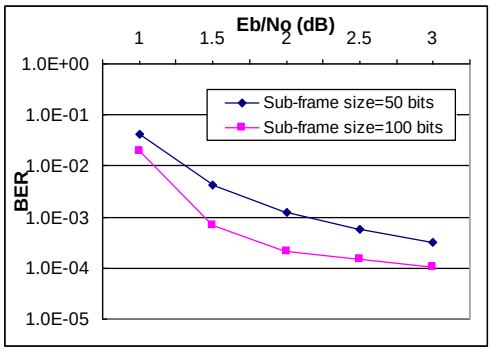


Figure 12. GPU-running results - frame length = 3000 bits, # of iterations = 5, WCDMA specification parameters.

Finally, the third approach is easier to apply than the first and second ones. Since the shared memory requirement is directly proportional to sub-frame length, as shown in Eq. (15), we can reduce the size requirement by shortening the sub-frame length (the sub-frame length is not specified by standards). But this approach comes at the cost of BER performance degradation. To show the BER performance trade-off, we measure the performance at two different sub-frame length = 50 bits and 100 bits, chosen based on the previous work [36]. As shown in Fig. 12, the 50-bit case shows about 0.5 dB coding gain loss at BER = 10^{-3} compared to the 100-bit case. Whether or not this loss is tolerable is determined by power-BER requirements. We fix the sub-frame length as 50 bits.

2.3 Experimental results – design space exploration

2.3.1 Design space exploration with the three axes

In this sub-section, we explore the design space of the Turbo decoding algorithm with the three axes: the radix-2 or 4, the BrM- or StM-centric mapping, and the number of sub-frames per thread block; we compare the performance of all possible cases and identify factors that differentiate the performance. We consider an even number of sub-frames per block from one to eight including one, for a total of 20 cases. A shared memory requirement of more than eight sub-frames exceeds the given resource in the ION (16 kB).

Table 1. Three factors to determine the occupancy.

# of subframes per block		1	2	4	6	8
# of threads per thread block	R2+StM	8	16	32	48	64
	R2+BrM	16	32	64	96	128
	R4+StM	8	16	32	48	64
	R4+BrM	32	64	128	192	256
Shared mem. req. per thread block (kB)	R2+StM	1.8	3.6	7.3	10.9	14.6
	R2+BrM	1.9	3.7	7.5	11.3	15.1
	R4+StM	1.1	2.3	4.6	6.9	9.2
	R4+BrM	1.3	2.5	5.1	7.7	10.2
Register req. per thread	R2+StM	22	22	22	22	22
	R2+BrM	21	21	21	21	21
	R4+StM	37	37	37	37	37
	R4+BrM	32	32	32	32	32

Shaded boxes: the number of threads is not an integer multiple of 32 including less than 32

In these experiments, we consider the WCDMA Turbo coding specification [35] with sub-frame length = 50 bits and frame length = 3,000 bits. A random interleaving is used.

We measure three parameters to determine the occupancy, as shown in Table 1. Within each combination of the radix and the mapping method, the increasing number of sub-frames increases the number of threads, but also increases the shared memory and register requirements per thread block. (The register requirement in Table 1 is per thread.) As a result, the occupancy does not monotonically increase or decrease, as shown in Fig. 13. Across the four combinations, the radix-4+BrM-centric combination shows the highest occupancy overall because it has benefits of both more number of threads (due to the BrM-centric) and the reduced shared memory requirement (due to the radix-4). The occupancies of the other three combinations are very low, mainly because of the large shared memory or register requirements. The cases with an unfilled warp, i.e., the number of threads per thread block is not an integer multiple of 32 including less than 32,

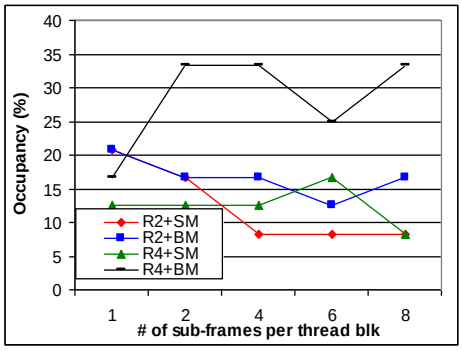


Figure 13. Occupancy.

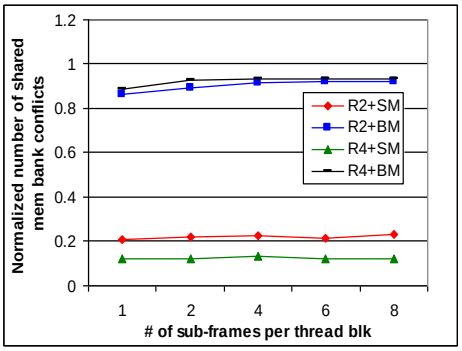


Figure 14. Normalized # of bank conflicts.

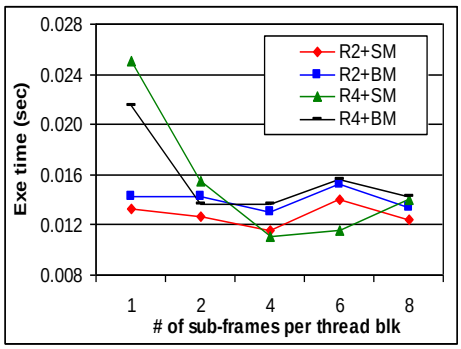


Figure 15. Execution time.

are displayed as the shaded boxes in Table 1. In addition to the occupancy, the warp fullness also affects performance.

To check whether or not device memory access is a reason for performance differences, we measure the number of coalesced/uncoalesced global memory references using the Ocelot tool [37]. The results show that all 20 cases have an exactly equal number of memory references – 30,000 (this number can be also calculated by Eq. (14)). We also measure the hit ratio of the constant cache – above 99% in all 20 cases. In addition, the local memory is not used at all in all 20 cases. Therefore, we conclude that device memory accesses do not contribute to performance differences in the design space.

Figure 14 shows the number of shared memory bank conflicts. The number is very different across the four combinations – both radix algorithms with the StM-centric

method show fewer bank conflicts than the other two with the BrM-centric method. The number of sub-frames per thread block does not have a considerable effect on shared memory bank conflicts (as described in Section 2.2.5.2).

Figure 15 shows the execution time for decoding one total frame. First, within each combination, the occupancy determines performance patterns when a warp is full, i.e., when the number of threads per thread block is an integer multiple of 32. In the R2+StM, the performance at the number of sub-frames per block = 4 and 8 is almost same due to the same occupancy. In the R2+BrM, the performance pattern is almost similar to the occupancy pattern except for the number of sub-frames per block = 1. In the R4+StM, the performance at the number of sub-frames per block = 4 is better than that at 8 due to the better occupancy. Finally, the R4+BrM is the combination which the performance pattern is best matched with the occupancy pattern because a warp is full at all five cases of the number of sub-frames per block.

Across the four combinations, the most critical factor to differentiate performance is shared memory bank conflicts. As shown in Fig. 14, the R4+StM combination shows the fewest number of bank conflicts, so it provides the best performance overall.

In summary, the two factors – the occupancy and the shared memory bank conflict – cause the performance difference in the design space.

2.3.2 Further performance improvements

The design space exploration in Section 2.3.1 was highly affected by the GPU parameters. In this section, we solely focus on the Turbo decoding algorithm for further performance improvements. According to our design space exploration, experiments in this section start with a configuration of radix-4, StM-centric mapping, and four sub-frames per thread block (the best performance case). After finally optimizing performance, we identify a performance bottleneck. (In Section 2.3.1, we did not identify performance bottlenecks, but found factors to cause the performance difference.)

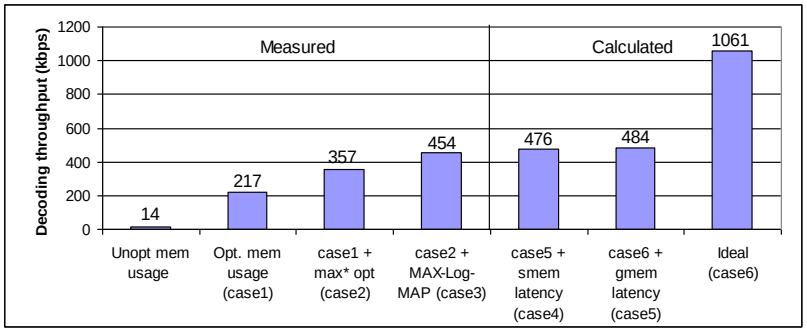


Figure 16. Performance optimizations on the ION GPU.

To improve decoding throughput further, we identify factors to limit attainable throughput in an algorithm aspect. The most critical factor is data dependencies existing during forward and backward recursions. Since the data dependency is an inherent nature of MAP-based algorithms, we cannot remove it without changing the algorithm itself. Instead, we can reduce the span of the data dependency by applying the trellis compaction (a transformation from the radix-2 to the radix-4) and by reducing the sub-frame length, which were discussed in the previous sections. The second critical factor is a max* operation, a kernel operation in the Log-MAP algorithm that is similar to an add-compare-select (ACS) kernel operation in the Viterbi algorithm except for the additional correction term. Since the number of max* calls is huge – every single alpha, beta, and LLR computation calls the max* operation, even trivial optimization for this operation can result in meaningful improvements of decoding throughput. The kernel operation can be optimized in different ways: by max* approximation (MAX-Log-MAP) [32], by using a lookup table, or by using faster intrinsic functions supported by underlying hardware platforms. Performance advantages by all three methods come at the expense of BER degradation. We apply the first and the last ones. For the last one, the ION GPU supports __logf(x) and __expf(x) [29] that are faster versions of the main element operations – log(x) and exp(x) – of max*.

Experimental results, after applying these optimizations, are shown in Fig. 16. A starting point is the case of R4+StM with four sub-frames per thread block (case 1). To

show performance improvements by memory usage optimizations, the performance of the case with unoptimized memory usage is also presented. In case 2, __logf and __expf are applied. In case 3, the MAX-Log-MAP algorithm is applied.

With the finally optimized version (case 3), we estimate throughput at additional three cases on the ION GPU. Case 6 in Fig. 16 assumes that access latency of all kinds of memories is zero – this is a superset of no shared memory bank conflicts, perfect global memory access coalescing, 100% hit ratio of constant cache – and the synchronization cost is also zero. We define this throughput as maximum attainable throughput on the ION GPU. Case 5 in Fig. 16 adds real global memory access latency, including coalescing effects, to case 6. Case 4 adds real shared memory access latency, including bank conflict-free effects, to case 5.

For estimating throughput at cases 4, 5, and 6, we count the number of executions of every instruction used in the case 3 program using the Ocelot tool [37]. Then we calculate the total number of required cycles to execute all the instructions and calculate throughput [in bps] as follows:

$$\textit{Total \# of required cycles per warp} = \sum_i \{(\textit{nominal CPI of inst\_warp}_i) \times (\textit{\# of inst\_warp}_i)\} +$$

$$(\textit{\# of gmem inst\_th per warp}) \times (\textit{gmem access latency per reference}) / (\textit{coalescing factor}) +$$

$$(\textit{\# of smem inst\_th per warp}) \times (\textit{smem access latency per reference}) / (\textit{bank conflict-free factor}). \quad (16)$$

$$\textit{Throughput} = (\textit{\# of information bits}) / \{(\textit{total \# of required cycles per warp}) \times (\textit{\# of warps per block}) \times (\textit{\# of blocks per SM}) / (\textit{multiprocessor clock frequency})\}. \quad (17)$$

Equations (16) and (17) are valid only when the number of threads per thread block is an integer multiple of 32. In Eq. (16), *inst_warp* is a warp instruction and *inst_th* is a thread instruction. The *gmem* and *smem* indicate the global memory and the shared memory, respectively. The *coalescing factor* and the *bank conflict-free factor* range from

Table 2. Comparisons with previous programmable implementations.

		Algorithm parameters			Machine parameters						Result
		Dec. algori-thm	Frame length	# of iter.	VLIW width	SIMD width	# of multi-proc.	Multi-proc. clock (GHz)	Mem. BW (GB/s)	Peak FLOPS (Gflops /s)	Throu-ghput (bits/s)
Our curre-nt work	ION	Max-log	5000	5	x	8	2	1.1	25.6	35	454 k
	Geforce 8600GTS	Max-log	5000	5	x	8	4	1.45	32	139.2	582 k
	Geforce 8800GTS512	Max-log	5000	5	x	8	16	1.63	64	416	1.15 M
	Tesla C1060	Max-log	5000	5	x	8	30	1.3	102	933	2.1 M
Previ-ous work	TI's C6X [18]	Max-log	N.A.	5	8	x	1	0.2	N.A.	> 1	57.2 k
	Starcore SC140 [19]	Max-log	5114	5	6	N.A.[1)]	1	0.3	N.A.	N.A.	1.88 M
	ST120 [19]	Max-log	5114	5	4	2[1)]	1	0.2	N.A.	N.A.	540 k
	TigerSharc[2)] [19]	Log	5114	5	4	N.A.[1)]	1	0.18	N.A.	0.9	666 k
	SODA [1]	Max-log	5114	5	x	32	1	0.4	N.A.	N.A.	2 M

1) In these three cores, each component instruction in a VLIW instruction supports SIMD operations. Except for ST120, the SIMD width is not clearly announced.
2) TigerSharc supports a dedicated instruction for the max* operation.

one to 16, respectively. The second and third terms (*gmem*-related and *smem*-related ones) in Eq. (16) are zero when calculating the throughput at case 6. We ignore the effects of the warp divergence and the constant memory access on performance according to the analysis in Section 2.3.1.

As shown in Fig. 16, the throughput at case 5 dramatically reduces from case 6. A reduction in throughput by the shared memory access (from case 5 to case 4) is very small. Therefore, we can conclude that global memory access latency is a major performance bottleneck at case 3 (the most optimized case). (In other cases, major performance bottlenecks may be different.)

The best optimized performance (case 3) is compared with the previous work that uses other programmable platforms in Table 2. In addition to the ION GPU, we measure performance on three additional GPUs – Geforce 8600GTS, Geforce 8800GTS 512, and Tesla C1060. The performance on all four GPUs is overall comparable to that of the other

programmable platforms except for the Starcore SC140 and SODA platforms; these two platforms are more customized to DSP applications than the GPUs. Especially, the SODA platform supports a customized instruction set and a specialized permutation data network for DSP applications.

To improve decoding throughput further, several additional supports can be considered on the GPU: first, putting more multiprocessors in the mobile GPU is most helpful and straightforward. Second, like other contemporary DSP cores, special hardware and instruction set supports for the max* kernel operation would result in considerable performance improvements without compromising BER performance. Finally, since some metrics in the Turbo decoding algorithm - such as alpha, beta, and gamma - do not need a full 32-bit word length [38], complete supports for 8-bit or 16-bit floating points may be also helpful.

The measured throughput in the baseline GPU architecture is 400 kbps and 418 kbps at # of sub-frames = 4 and 8 respectively, which are much lower than the WCDMA 2 Mbps requirement. For further performance improvements, we propose architectural enhancements to existing GPUs in the following sections.

2.4 Proposed architectural enhancements

In the previous sections, we explored the design space of the Turbo decoding algorithm on GPUs and identified a performance bottleneck.

In this section, we propose architectural enhancements to existing GPUs to increase throughput of embedded applications – especially, the Turbo decoding algorithm. We try four categories of architectural enhancements: increasing shared memory resources to raise the occupancy, supporting a sub-word parallelism, adding special instructions/hardware for data and memory index computations, and increasing the number of streaming multiprocessors.

2.4.1 Shared memory resources (size)

According to the baseline experimental results, the low occupancy – 8% at most – is one of the major factors to restrict performance. A higher occupancy is related to a larger pool of ready warps available. A ready warp can be chosen from the pool for execution when the current warp stalls mainly because of global memory access latency.

Table 1 shows the three parameters to determine the occupancy, and the shared memory requirement turns out to be an occupancy-limiting factor.

In our implementation, the shared memory is allocated to alpha and beta metrics and some intermediate variables since they must be shared among threads of the same thread block. This requirement of data sharing originates from data permutations between trellis stages. Sharing of gamma metrics can be avoided by mapping BrM computations to threads according to our proposal (as described in Section 2.2.5). The intermediate variables occupy negligible space. Therefore, either alpha or beta metrics take up most of the shared memory. In our implementation, a forward recursion (alpha computations) is done first; then LLR values are calculated during a backward recursion consuming beta computation results right away – i.e., no need to store beta metrics. As a result, alpha metrics occupy nearly all the shared memory. An alpha memory requirement [in bits] per thread block is estimated as follows:

(*# of states per trellis stage*) x (*sub-frame length* [in bits]) x (*# of sub-frames per thread block*) x (*word length of an alpha metric* [in bits]), (18)

where the word length is 32 bits (a single-precision floating-point). An estimation result by Eq. (18) is almost equal to experimental results of Table 1. This size requirement affects the number of active warps per multi-threaded streaming multiprocessor, so it is eventually related to the occupancy.

According to this analysis, the first architectural enhancement is to add more shared memory resources to increase the occupancy.

2.4.2 Sub-word parallelism

In addition to the SIMD parallelism, a sub-word parallelism is another promising and cost-effective method to improve throughput in multimedia applications, including Turbo decoding.

External and internal data in MAP-based algorithms can be represented with 16 bits [38]. Two 16-bit sub-words are packed into one 32-bit word (currently a single-precision data type supported by NVIDIA GPUs). Computations for two sub-words can be executed at a time. Since only nine or 10 bits of 16 bits are enough to represent amplitudes of metrics in the Max-Log-MAP algorithm, a carry is not generated at the sub-word boundary. Therefore, sub-word-level computations can reuse existing resources for word-level computations. Even an AND logic, which is generally required in architectures to support the sub-word parallelism [39], is not needed. In addition, the existing instruction set can be used for sub-word parallel computations (though this is not described in this thesis).

2.4.3 Special instructions/hardware for data computations

According to the algorithm analysis in Section 2.2.1, we add two special instructions/hardware for data computations. The first one supports an ACS operation that is called every alpha, beta, and LLR computations. Another new instruction supports a branch metric computation.

ACS operations are classified into two categories according to the number of input arguments in a max operation: two arguments in alpha/beta computations, and eight arguments in an LLR computation.

A single ACS operation of alpha and beta computations consists of two add, one max, and three shared memory instructions and is completed in 24 cycles on the baseline GPU, as shown in Fig. 17 (each instruction is a warp instruction and there are no shared memory bank conflicts, so 6 insts x 4 cycles = 24 cycles). A horizontal dashed line

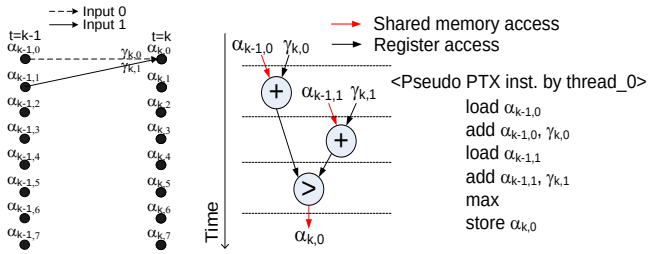


Figure 17. ACS operation in the baseline GPU.

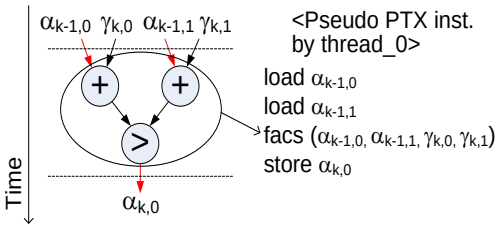


Figure 18. ACS operation in the advanced GPU.

indicates that arithmetic/logic operations separated by the line are executed by different instructions at different cycles. The dashed lines and red and black arrows in the following figures have the same meaning as in Fig. 17.

We add a special instruction, *facs* (fast ACS), to finish a single ACS operation in 16 cycles including the three memory instructions, as shown in Fig. 18. Three shared memory accesses are still serialized, but all arithmetic/logic operations are done by a single *facs* instruction. By assigning more hardware resources, including an additional adder, and/or by applying a fast addition algorithm (e.g., carry-look-ahead), a *facs* instruction can be completed in four cycles. To support *facs*, the number of read ports of the register file must be increased from two to four causing considerable area overheads. To remedy this problem, we add two temporary registers (not shown here) to hold data from the shared memory, instead of using the register file.

Since ACS operations related to different trellis states are mapped to different threads (i.e., different scalar processors), a special hardware unit for supporting *facs* must be added to every scalar processor (Figs. 17 and 18 are of the 0^{th} scalar processor as an example).

This savings of eight cycles per single ACS operation is accumulated as follows:

Total savings [*in cycles*] = 8 *cycles* x (*sub-frame length* [*in bits*]) x 2 (*due to forward and backward recursions per half-iterations*) x 2 (*due to two half-iterations per iteration*) x (*# of iterations*) x (*# of thread blocks per SM*). (19)

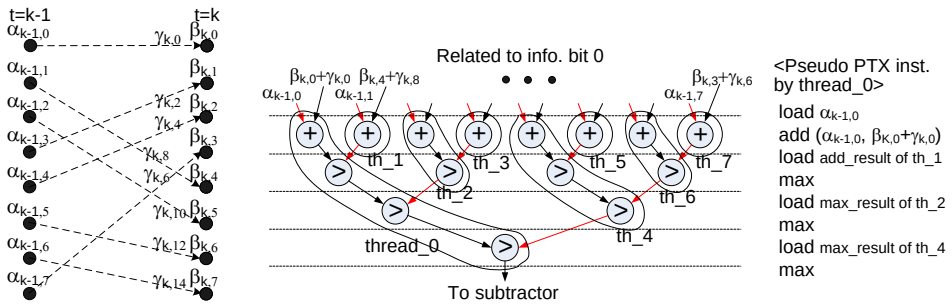


Figure 19. LLR computation related to an info bit '0' in the baseline GPU.

LLR computations require another category of an ACS operation with an eight-input max operation. In the baseline GPU, the eight-input max operation can be implemented by a tree structure of two-input max operations as shown in Fig. 19. The mapping method of LLR computations to threads, shown in Fig. 19, is an example. Our mapping method tries to minimize the frequency of shared memory access; particularly, by assigning beta (not shown here) and partial LLR computations of one branch to the same thread, previously generated beta+gamma results can be loaded from the registers instead of the shared memory. For example, thread_0 computes $\beta_{k,0}+\gamma_{k,0}$ during a $\beta_{k-1,0}$ computation related to the first branch, and stores the $\beta_{k,0}+\gamma_{k,0}$ to a register (this step is not shown in Fig. 19). To compute a LLR value, thread_0 loads the $\beta_{k,0}+\gamma_{k,0}$ result from the register (instead of the shared memory) and computes $\beta_{k,0}+\gamma_{k,0}+\alpha_{k-1,0}$ related to the same branch.

Note that another set of operations related to info bit '1,' which is identical to Fig. 19 except for different combinations of alpha and beta inputs, is executed in the following cycles. The number of required cycles is {(8 insts related to info. bit '0') + (8 insts related to info. bit '1')} x 4 = 64 cycles.

Principles behind the LLR computation mapping onto our enhanced architecture are resource sharing and shared memory access minimization. First, the newly added resources for a *facs* operation of alpha/ beta computations can be shared by LLR computations, as shown in Fig. 20. This resource sharing, however, introduces additional

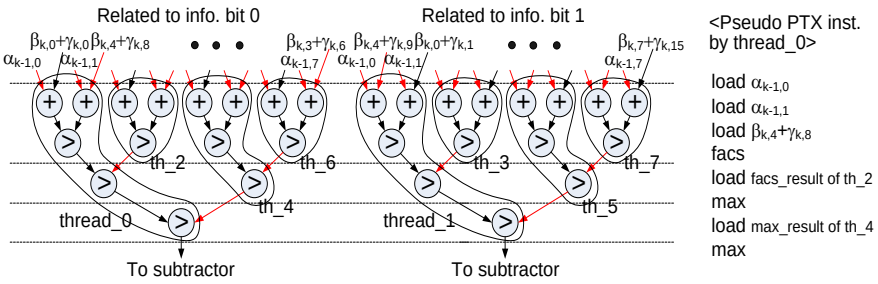


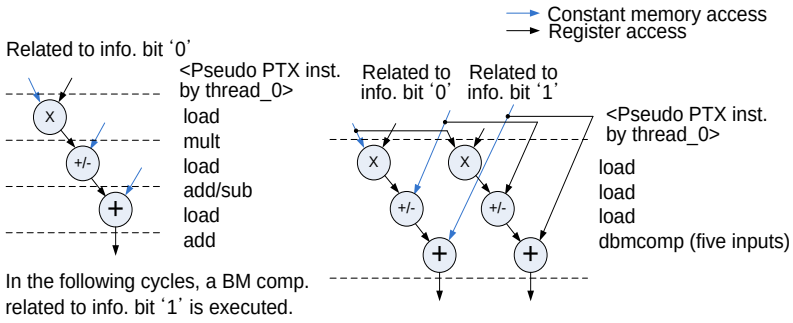
Figure 20. LLR computation in the enhanced GPU.

shared memory accesses for loading beta+gamma results. We try to minimize this additional shared memory access; by our mapping method, a half of the beta+gamma results can be loaded from registers, as shown in Fig. 20. The number of required cycles is 8x4 = 32 cycles, giving a 32-cycle savings per single LLR computation. This savings is also accumulated according to Eq. (19) except that the factor 2 now reduces to 1 since LLR computations are done only during backward recursions.

In the Max-Log-MAP algorithm, a BrM computation is different from alpha, beta, and LLR computations; it does not use an ACS kernel operation. Instead, it is the only computation using a multiplication operation. Therefore, a different special instruction is needed for BrM computations.

According to Eq. (3), a gamma computation requires two multiplications and two additions. Since u and u^p in Eq. (3) take a value of +1 or -1 respectively, a gamma computation can be optimized by a compiler, as shown in Fig. 21(a) (a different optimization is possible, but the nvcc compiler ver. 2.3 generates this sequence of instructions).

In the baseline GPU, a BrM computation takes {(6 insts related to info. bit '0') + (3 insts related to info. bit '1')} x 4 = 36 cycles (the three loads are not needed in the computation related to info. bit '1'). In the enhanced GPU, a new instruction called *dbmcomp* (double BrM computations) is supported to finish two BrM computations of a pair of branches – e.g., the two BrMs, $\gamma_{k,0}$ and $\gamma_{k,1}$, in Fig. 17 – in four cycles excluding



(a) In the baseline GPU **(b) In the enhanced GPU**

Figure 21. BrM computation in both GPUs.

loads from constant memory. Note that the number of read ports of a register file must be increased to five, which results in big area overheads. The temporary register solution used for *facs* is applied here as well. The number of required cycles is 16 cycles including the three loads, resulting in a 20-cycle savings. This savings is also accumulated according to Eq. (19).

2.4.4 Special instruction/hardware for memory index computations

In addition to data computations, memory index calculations are also a significant part of the CUDA code for the Turbo decoding algorithm. Especially, address calculations for constant memory loads are most critical. The shared memory is local to each SM, so its address calculation is simpler than that of the constant memory. The global memory is not frequently used in our implementation. For a more detailed description of memory allocations in our mapping method, we refer readers to [40].

We propose to add a special functional unit (SFU), which is separate from the SPs, for memory index computations. A related new special instruction is called *memidx*. Data computations and memory loads/stores on the SPs and memory index computations on the SFU are processed simultaneously in a pipelined way – i.e., the current result of a memory index computation done by the SFU is used for the data computation by the SPs in the next cycles.

2.4.5 Special instruction/hardware for data communications

The requirement of intensive data permutations is known as a major performance bottleneck in MAP-based Turbo decoding algorithms. Data permutations can be supported by strided memory access requiring multiple memory access ports or by a specific permutation network [41]. The baseline GPU takes a different approach to support data permutations – i.e. by constructing the shared memory in a multi-bank structure. The multi-bank structure does work when there are no bank conflicts. In our all proposed architectural enhancements, we ensure zero shared memory bank conflict. Therefore, a special instruction/hardware for data communications is not needed.

2.4.6 Number of SMs

Finally, increasing the number of multiprocessors is the most general (not specific to the Turbo decoding algorithm) and straightforward way to improve throughput. The baseline architecture (ION GPU) has two SMs that is less than other GPUs – e.g., eight or 16 in ION II, 16 in Geforce 8800GTS 512, and 30 in Tesla C1060. Therefore, increasing the number of SMs over two is a feasible solution.

Since a Turbo decoding application has a small-scale workload compared to other graphic and scientific applications, we should carefully determine the maximum number of SMs so that none of the SMs are in idle states. Furthermore, increasing the number of SMs without co-improving the memory system can worsen global memory latency by generating more memory traffic.

2.5 Evaluations

2.5.1 Experimental method

We use an in-house cycle-accurate and trace-driven simulator for our experiments. We generate traces of the turbo decoding algorithm for different inputs and configurations using GPUOcelot [37]. The traces are generated at a warp granularity rather than at a

thread granularity since a warp is the unit of execution in the GPU. Though traces are generated at a warp granularity, any control divergence exhibited by the threads of a warp is captured in the traces. All the memory addresses generated by the different threads of a warp for the same instruction are also included in the traces. In addition to modeling the multi-threaded pipeline of GPU cores (Streaming Multiprocessor), the simulator models the GPU memory hierarchy in considerable detail. Shared memory bank conflicts, constant cache, texture cache, DRAM bus congestion, DRAM bank conflicts and DRAM timing constraints are all modeled faithfully. The simulator also includes a fixed delay interconnect.

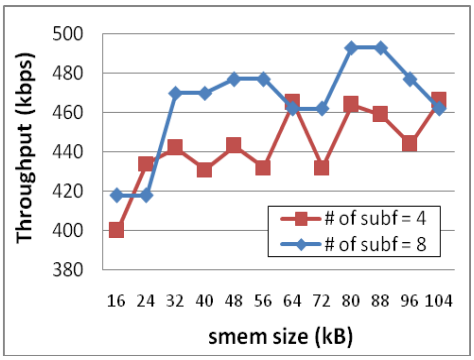
The simulator is configured to simulate the ION GPU and Turbo coding traces generated using GPUOcelot are fed to the simulator. First, we measure throughput on the baseline GPU architecture. Then we apply each architecture enhancement individually and show performance improvements in Section 2.5.2. We also show a trade-off between throughput and area overheads in Section 2.5.3.

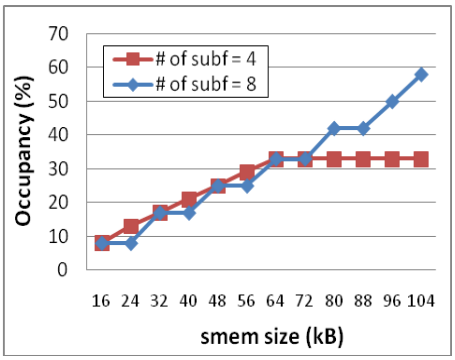
2.5.2 Throughput results

We measure a throughput improvement by each architecture enhancement separately – e.g., when measuring performance on the enhanced architecture with the sub-word parallelism support, other parameters such as shared memory resources and the number of SMs are set to the same values as in the baseline architecture, 16 KB and two SMs. Throughput on the baseline architecture is used as a reference performance; the reference performance is 400 kbps at # of sub-frames = 4 and 418 kbps at # of sub-frames = 8.

First, throughput is improved overall as we add more shared memory resources to increase the occupancy, as shown in Fig. 22 (a), but does not monotonically increase. Arithmetic intensity of the Turbo decoding algorithm is calculated as follows:

(*# of alpha, beta, gamma, and LLR computations per trellis stage*) / (*# of global memory access per trellis stage*). (20)





(a) Throughput **(b) Occupancy**

Figure 22. Effect of more shared memory resources (size) (Section 2.4.1).

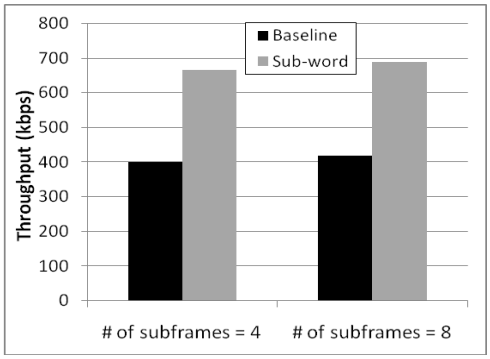


Figure 23. Effect of the sub-word parallelism (Section 2.4.2).

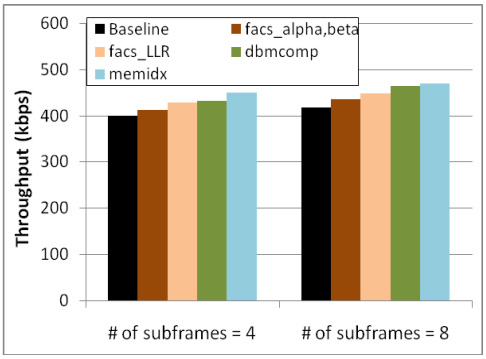


Figure 24. Effect of the special inst./hw (Sections 2.4.3 and 2.4.4).

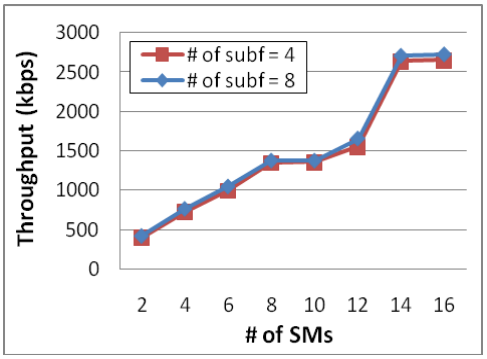


Figure 25. Effect of increasing # of SMs (Section 2.4.6) .

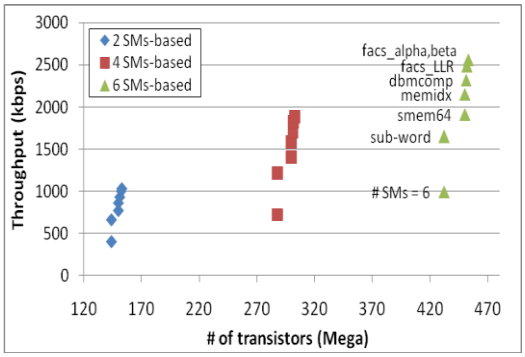


Figure 26. Throughput vs. area (# of subframes=4).

In our implementation, the numerator of Eq. (20) is 8 + 8 + 16 x 2 + 1 = 49 and the denominator is 1 (for storing an extrinsic value). Because of this high arithmetic intensity, the effect of the occupancy is saturated at some points – the optimal memory

size resulting in highest throughput is 64 kB at # of sub-frames = 4 and 80 kB at # of sub-frames = 8.

Figure 23 shows a throughput improvement by supporting the sub-word parallelism. Almost a 70% improvement is achieved by doubling the execution width. Figure 24 shows throughput improvements by applying each new instruction/hardware one by one. The improvements by supporting *facs* for both alpha/beta and LLR computations is very small – 3% by alpha and beta *facs* and 7% by LLR *facs*. The improvements by *dbmcomp* and *memidx* instructions are greater than them by *facs*; this difference is caused by a difference number of arithmetic/logic instructions constituting a computation. In other words, optimization space for branch metrics and memory index computations is wider than that for alpha/beta and LLR computations. Finally, Fig. 25 shows performance improvements by increasing the number of SMs. Along with the sub-word parallelism, increasing the number of SMs is the most effective approach to improve throughput. The experimental results demonstrate that without any hardware enhancements, to reach the 2 Mbps requirement, we need more than 12 SMs, which will increase area overheads significantly. We can reduce the number of required SMs significantly with our proposed architectural enhancements as described in Section 2.5.3.5.

2.5.3 Area overheads

In this section, we calculate area overheads caused by our architectural enhancements in terms of the number of transistors.

2.5.3.1 Adding shared memory resources (size)

The shared memory is SRAM, which requires six transistors per bit. The optimal memory size, 64 kB and 80 kB (see Fig. 22(a)), requires about 3 M and 4 M transistors respectively. This number is multiplied by # of SMs to get the total number of transistors that should be added.

2.5.3.2 Sub-word parallelism support

As described in the previous section, supporting the sub-word parallelism does not require additional hardware resources. Even if a carry is generated at the sub-word boundary, only an AND gate is required to either pass (in the normal mode) or block (in the sub-word mode) the carry. Its area overhead is negligible.

2.5.3.3 Special instructions/hardware

The *facs* instruction requires one additional adder, *dbmcmp* requires one additional multiplier and two additional adders as shown in Figs. 18 and 21(b), and *memidx* requires one adder and one multiplier (not shown in this thesis). These additional components for *facs* and *dbmcmp* support 32-bit floating-point computations, and the components for *memidx* support 32-bit integer computations. The number of transistors of the 32-bit floating-point adder and multiplier are about 2,400 and 28,000 [42] respectively. The number of transistors of the 32-bit integer adder and multiplier are about 768 [43] and 25,232 [44] respectively; these numbers are rough estimates (e.g., these numbers are different depending on addition and multiplication algorithms applied). Since the order of the number of transistors required for shared memory resources and SMs is in millions, the rough estimates are acceptable for our area estimate purpose. These new components must be added to every SP, so the total number of additional transistors is calculated as follows:

(*facs_trans* + *dbmcmp_trans* + *memidx_trans*) x 8 (due to # *of SPs per SM*) x (# *of SMs*), (21)

where **_trans* is the number of transistors required for the * operation.

2.5.3.4 Increasing the number of SMs

Although increasing the number of SMs is the most efficient way to improve throughput, it has the biggest area overhead among all architectural enhancements.

The number of transistors per SM can be estimated as 72 M from the datasheet of Geforce 8600 GTS [45]. Please note that the number of transistors is typically much less in GPUs for mobile platforms, so our estimation provides an upper bound.

2.5.3.5 Putting them all together – throughput vs. area

Figure 26 shows the trade-off between throughput and area overheads. All architectural enhancements are put together. The order in which the architecture enhancements are applied is determined by percentage improvements – the bigger, the earlier. So # of SMs is first applied, and *facs_alpha,beta* is finally applied – the sequence is indicated beside the 6 SMs-based case in Fig. 26 and it is same for the other two cases.

As shown in Fig. 26, by applying all the enhancements on top of four SMs, we can achieve 1.9 Mbps almost close to the WCDMA requirement. By applying the sub-word parallelism, 64 kB shared memory, and *memidx* instruction on top of six SMs, we can achieve 2.2 Mbps throughput. Please note that without any enhancements, we need at least 12 SMs (Fig. 25), which uses 864 M transistors. If area is given as a design parameter, we should consider the trade-off between throughput and area when selecting some of the architectural enhancements.

2.6 Related work

Some previous work implemented the Turbo decoding algorithm in industrial DSP cores such as TI's C6x and Starcore SC140 [3, 4]. Unfortunately, their throughput is below the standards' requirements. For example, none of them satisfy the 2 Mbps requirement of the WCDMA standard.

On the academic side, Lin et al. [41] proposed a more specialized multimedia architecture called SODA. By supporting some specialized instructions and a permutation network on top of the SIMD+VLIW-style architecture, it is possible for SODA to satisfy

the WCDMA throughput requirement [46]. It is the only programmable platform to do so, but it is not yet available commercially.

Our goal is to achieve the 2 Mbps requirement on a commercially available programmable core, the GPU. Turbo decoding is used in mobile systems where small area and low energy consumption are required. Therefore, we choose the ION GPU, one of the lightweight chips, as the baseline architecture. We propose architectural enhancements to it.

Contributions of our work are as follows:

1. We map the Turbo decoding algorithm onto the GPU according to the result of parallelism analysis. We explore the design space of the Turbo decoding algorithm on the GPU with three major axes.

2. We enhance the baseline GPU architecture to improve throughput of Turbo decoding above the WCDMA requirement.

2.7 Summary

We explored the design space of the Turbo decoding algorithm on GPUs. We considered three axes for the exploration: the radix degree, the mapping method, and the number of sub-frames per thread block. In the Turbo decoding algorithm, combined effects of the algorithmic and implementation aspects make the design space exploration much more complicated.

The experimental results demonstrate that the radix-4 algorithm with the StM-centric mapping method provides the best performance at four sub-frames per thread block. According to our analysis, the occupancy and shared memory bank conflicts are major factors to differentiate the performance in the design space. We further improved performance by optimizing the kernel operation and applying the MAX-Log-MAP algorithm. At the finally optimized case, the global memory access is a major performance bottleneck.

Even with the best optimized case, the performance does not satisfy the WCDMA requirement. We proposed architectural enhancements to GPUs for increasing throughput. We tried four categories of architectural enhancements: increasing shared memory resources to raise the occupancy, supporting the sub-word parallelism, adding special instructions/hardware for data and memory index computations, and increasing the number of streaming multiprocessors. Experimental results by our cycle-accurate GPU simulator demonstrate that increasing the number of SMs and supporting the sub-word parallelism are the most promising ways to improve throughput. With the proposed enhancements, we achieved the WCDMA throughput requirement with only four to six SMs, while without any enhancements at least 12 SMs are required. When an area limitation is given as a design parameter, we should select some of the architectural enhancements based on our throughput-area trade-off analysis.

CHAPTER III

OFF-CHIP DRAM MAIN MEMORY SYSTEMS

Our final goal related to DRAM systems is to improve system throughput by exploiting useful features of DRAM systems in DSP multiprocessor systems. We first represent DSP applications as data flow graphs and allocate a part of DRAM main memory to some graph edges (called *buffer mapping*) for communication and synchronization between processing elements.

We first introduce DRAM system fundamentals and graphical representations of DSP applications. And then we explain DRAM usage requirements of DSP applications. Finally, we propose two high-performance buffer mapping methods that exploit the bank concurrency of DRAM main memory systems.

3.1 DRAM system fundamentals

In this section, we describe the contemporary DRAM main memory system and its useful features. Figure 27 depicts the contemporary DRAM main memory system including a

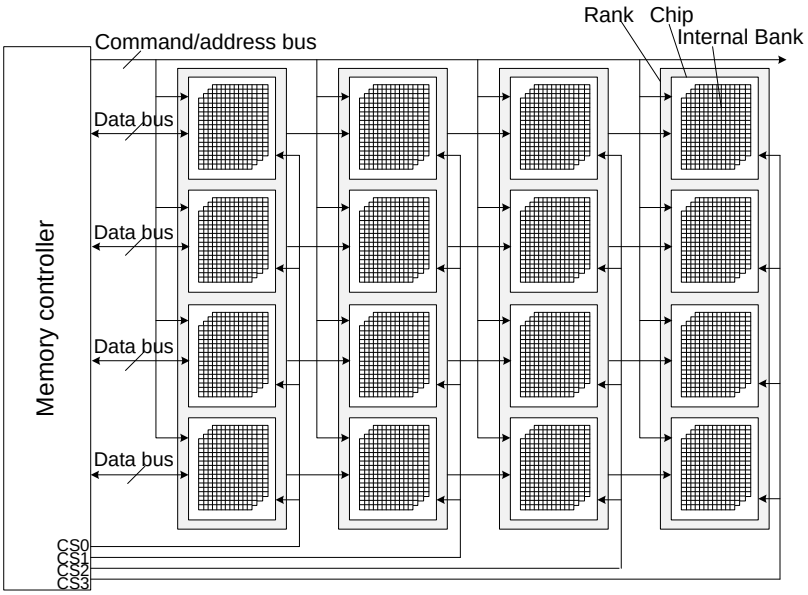


Figure 27. Contemporary DRAM main memory system.

memory controller. The system consists of several ranks, each rank has several chips, and each chip has several internal banks. The multiple internal banks within a chip and the multiple ranks provide multiple levels of concurrency. Note, however, that the multiple chips within a rank are not for providing concurrency, but for providing wide data transfers. In the example of Fig. 27, the number of ranks, the number of chips, and the number of internal banks are all four. We consider a single-channel SDRAM main memory system, where all banks and ranks share a single address/command bus and a single data bus as shown in Fig. 27. Furthermore, we assume that a commodity SDRAM chip with a relatively low frequency (e.g., see [47, 48]) is used as a component chip – i.e., each bank capacity and the number of banks per chip are fixed, but we can adjust the numbers of chips and ranks according to desired configurations.

The DRAM memory system has two useful features: bank concurrency and page mode – data accesses to different banks and data accesses to the open row result in low access latency. Bank concurrency makes it possible to hide the latency caused by precharge and row activation commands. We use the terms command and transaction when describing main memory operations. A command is issued by the memory controller to the DRAM. Examples of operations referenced in a command are precharge, row activation, and column access. On the other hand, a transaction refers to an interaction between the processor and the memory controller. Examples of transactions are load and store operations. In general, a single transaction generates one or more commands depending on the row buffer management policy and the transaction schedule. Since an address/command bus and a data bus are shared among all banks in a single-channel memory system, parallel execution of commands is actually done in a pipelined fashion. Second, page mode is used to exploit temporal and spatial row locality of DRAM memory accesses. Each internal bank has its own row buffer. When the current memory transaction goes to the same row as the previous memory transaction, the current transaction gets the data from the open row buffer, not from the bank itself.

Several techniques [13, 15, 16], which utilize both bank concurrency and page mode, exist to reorder memory commands in a memory controller. We focus in this thesis on exploiting one of them – bank concurrency – carefully, and on managing our exploitation systematically through a high-level compiler. For analyzing the pure effect of bank concurrency, we need to eliminate the effect of page mode on system performance. Therefore, the close page policy is selected as the row buffer management policy in this thesis. Under the close page policy, every transaction is converted to a sequence of precharge → row activation → column access commands regardless of row hit/miss status, so we can eliminate the effect of page mode on system performance.

If we use a multi-rank configuration as in Fig. 27, rank concurrency is also used in addition to bank concurrency. In this thesis, the term bank concurrency indicates the concurrency among the banks within each chip. The term rank concurrency indicates the concurrency among the banks across the rank.

3.2 Graph representations of DSP applications in multi-core systems

3.2.1 SDF and IPC graphs

Dataflow graphs provide a natural representation format for DSP applications [49]. Synchronous dataflow (SDF) is a special case of dataflow in which the number of tokens consumed or produced by each actor (computational task) in each firing (task execution) is known a priori [50]. An important class of DSP applications exhibits this form of synchrony, which allows compilers to perform more powerful scheduling and buffer mapping optimizations compared to more general models of computation (e.g., see [51, 52]). This results in reduced run-time overhead, streamlined buffer memory requirements, and more predictable run-time behavior.

To help analyze system performance, an inter-processor communication (IPC) graph can be generated from an SDF application graph and a self-timed scheduling result for the graph. The IPC graph models self-timed execution of the given application based on

the given schedule, and explicitly shows requirements of inter-processor communication and synchronization [49]. In the shared-memory programming model, communication is implicitly performed by memory read and write transactions, but synchronization requires an explicit mechanism. In embedded computing systems, software synchronization mechanisms based on hardware primitives, which are widely used in general-purpose computing systems, are often undesirable because of their hardware costs. Instead, the bounded buffer synchronization (BBS) and unbounded buffer synchronization (UBS) protocols are considered, which use the shared main memory for synchronization as well as communication. The BBS protocol is more attractive because of its bounded buffer feature.

3.2.2 BBS synchronization protocol and buffer mapping

In this section, we describe the BBS protocol and related buffer mapping considerations.

Figure 28 depicts a SDF application graph and an associated scheduling result on three processors. Note that this scheduling result is in general not unique - it is determined by one of many scheduling possibilities [49]; the one illustrated here is a represented result that we have chosen for the purpose of illustration. Every actor is indexed with a unique number and this number is used to identify the actor in the schedule.

The scheduling result shown here is derived by using the classic HLFET algorithm [53] under the self-timed paradigm. In this thesis, we consider a homogeneous SDF (HSDF) graph [50]. Since all arbitrary SDF graphs can be converted into equivalent HSDF graphs [50], the techniques of this thesis are also applicable to general SDF graphs, as long as the SDF-to-HSDF transformation is applied appropriately as a preprocessing step.

From the SDF application graph and the schedule, we obtain the IPC graph shown in Fig. 29. In the IPC graph, each white circle represents a computation actor, and each gray

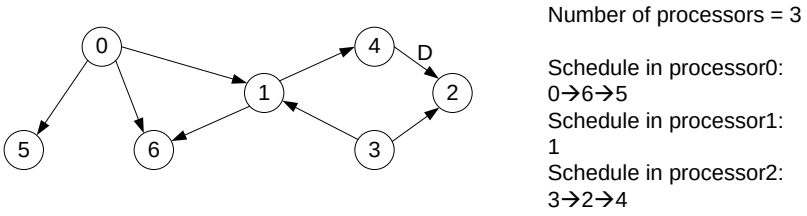


Figure 28. SDF application graph along with a schedule for the graph.

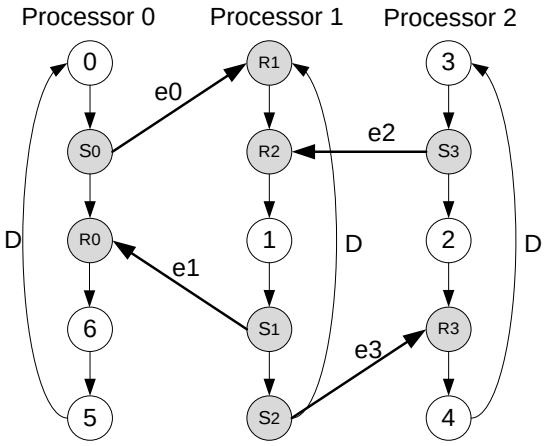


Figure 29. IPC graph of the SDF graph in Fig. 28.

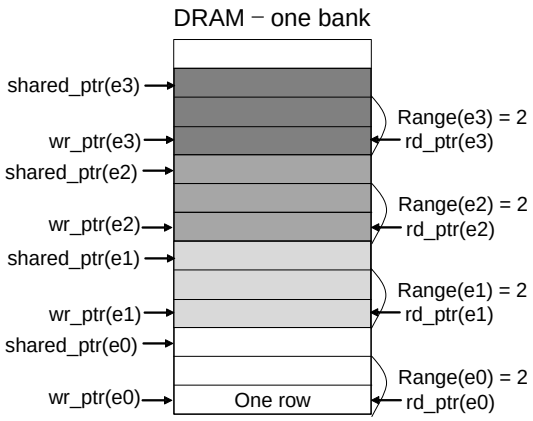


Figure 30. Sequential buffer mapping.

circle represents a communication actor. Communication actors are labeled as (S) or (R) to represent inter-processor *send* and *receive* operations, respectively. The actors that are assigned to the same processor by the given schedule are connected together so that they form a cycle. The ordering of actors along each of these cycles corresponds to the actor ordering for the corresponding processor in the self-timed schedule. Each of these cycles represents the iterative, sequential execution of the subset of actors that is assigned to a given processor.

For each edge in the SDF application graph whose source and sink actors are assigned to different processors by the given schedule, IPC edge is instantiated between the associated send and receive actors in the IPC graph. For example, there are four IPC edges, *e*0-*e*3 in Fig. 29. For further details on IPC graph construction, we refer the reader to [49].

Suppose that we are given an IPC edge e which is a feedback edge, and let $src(e)$ and $snk(e)$ denote, respectively, the source and sink actors of e. Figure 30 depicts the sequential buffer mapping for the IPC graph of Fig. 29 under the BBS protocol. For simplicity, we suppose the transferred data size is one row. In the BBS protocol [54], a write pointer $wrptr(e)$ for e is maintained on the processor that executes $src(e)$, a read pointer $rdptr(e)$ for e is maintained on the processor that executes $snk(e)$, and a copy of $wrptr(e)$ is maintained in some shared memory location $shptr(e)$. The pointers $rdptr(e)$ and $wrptr(e)$ are initialized to zero and $del(e)$, respectively, where $del(e)$ denotes the logical delay (number of initial tokens) on e.

Just after each execution of $src(e)$, the new data value produced onto e is written into the shared memory buffer for e at offset $wrptr(e)$, and $wrptr(e)$ is updated by the following operation

$$wrptr(e) \leftarrow (wrptr(e) + 1) \bmod range(e). \tag{22}$$

Furthermore, $shptr(e)$ is updated to contain the new value of $wrptr(e)$. Just before each execution of $snk(e)$, the value contained in $shptr(e)$ is repeatedly examined until it is found to be not equal to $rdptr(e)$. Then, the data value residing at offset $rdptr(e)$ of the shared memory buffer for e is read, and $rdptr(e)$ is updated by the operation

$$rdptr(e) \leftarrow (rdptr(e) + 1) \bmod range(e). \tag{23}$$

Since all IPC edges in our proposed methodology can be assumed to be feedback edges, the size of each buffer is known at compile-time, so we can determine the start address and the range of each buffer at compile-time. Only the modulo-based increase of the pointers needs to be carried out during run-time.

The sequential buffer mapping of Fig. 30 is the most straightforward, but does not utilize useful features of the contemporary DRAM main memory system.

3.3 Proposed buffer mapping methods

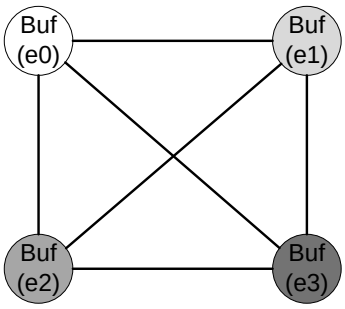
In Section 3.2, we represented DSP applications as SDF/IPC graphs and described DRAM space requirements of DSP applications in multiprocessor DSP systems.

In this section, we propose two high-performance buffer mapping methods that use two general approaches to improve throughput: *parallelism* and *speculation*.

If applications show opportunities of concurrent executions, application throughput can be improved by supporting concurrent hardware resources. In this thesis, the opportunities in applications exist in comm/sync transactions of several IPC edges and the concurrent hardware resources are DRAM banks. IPC edges of an IPC graph indicate comm/sync requirements between processors. In the shared-memory programming model with the BBS synchronization protocol, a part of main memory space should be allocated to each IPC edge to perform communication and synchronization. If comm/sync transactions of several IPC edges occur in an overlapped time slot, throughput can be improved by assigning a unique DRAM bank to each IPC edge.

The first proposed mapping method only exploits one type of parallelism – IPC edge-level parallelism. The second one uses speculation on the top of the first mapping method to further exploit another type of parallelism – comm/sync-level parallelism. The two kinds of parallelisms have a hierarchical structure – i.e., comm/sync-level parallelism is a subset of IPC edge-level parallelism.

Although mapping methods to consider both balanced mapping and balanced access are the best, our mapping methods focus on the balanced mapping of IPC buffers. It can be more difficult to achieve balanced access in systems using the BBS protocol. In the BBS protocol, both communication and synchronization are performed by memory transactions to the shared main memory. If a specific receive-actor tries a sync_read before the corresponding send-actor completes its associated write and sync_write, then the sync_read check fails, and it must be retried until the sync_read check succeeds. This generally generates more accesses to the specific IPC buffer corresponding to the IPC edge that has the synchronization check failure. So in systems using the BBS protocol, the degree to which accesses are balanced is strongly affected by the rate of synchronization failures and associated needs to reattempt synchronization operations.



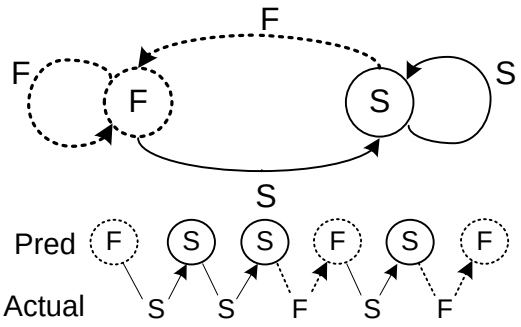


Figure 31. Interference graph of IPC buffers. Figure 32. 1-bit predictor of sync_read results.

Further research on mapping methods to consider balanced access might be a good direction for future work.

3.3.1 First proposed buffer mapping method

To analyze the edge-level parallelism, an interference graph of IPC edges is generated by using a graph coloring technique. In an interference graph, each vertex represents an IPC buffer and each edge between vertices represents interference between the associated buffers. That is, two vertices are connected with an edge if the corresponding two buffers are accessed in overlapping segments of time. To facilitate exploitation of the parallelism, the two buffers should be mapped to different banks. The interference graph of the IPC graph of Fig. 29 is shown in Fig. 31. According to interference analysis based on many simulations for this example, all of the four buffers turn out to exhibit interference, so the four buffers should all be mapped to different banks.

3.3.2 Second proposed buffer mapping method

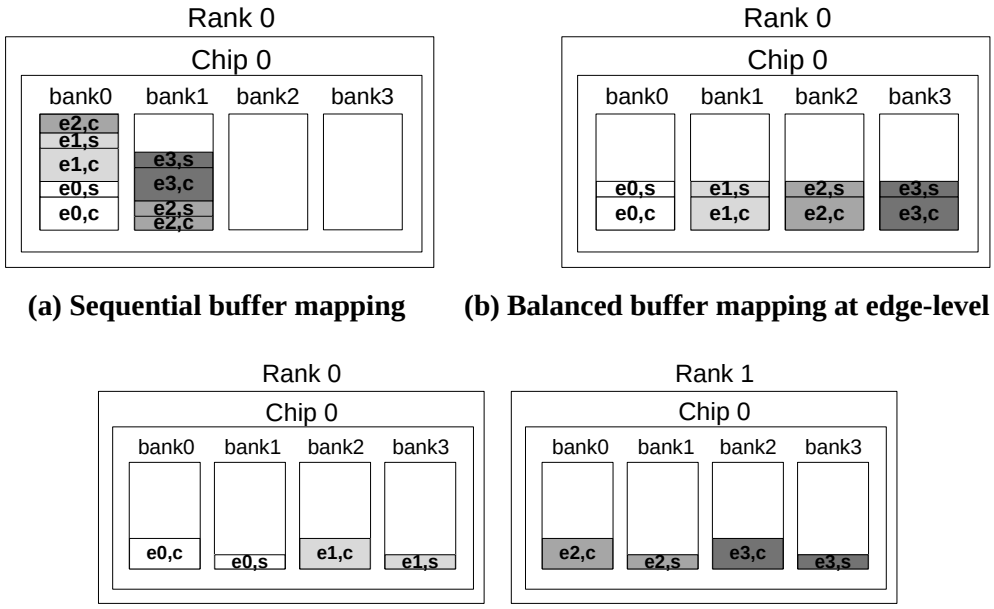
To exploit DRAM concurrency more, we use an additional form of parallelism at the comm/sync-level within each IPC edge. Every IPC buffer consists of two kinds of transaction buffers: a communication transaction buffer and a synchronization transaction buffer. In the balanced mapping using the edge-level parallelism, communication and synchronization buffers of an IPC buffer are mapped to the same bank. In contrast, in the

balanced mapping using the comm/sync-level parallelism, communication and synchronization buffers are mapped to different banks.

To analyze the comm/sync-level parallelism, we use the 1-bit state machine as shown in Fig. 32 to predict results of sync_read operations for each IPC edge (every IPC edge has its own state machine). In Fig. 32, `F' and `S' mean “failure” and “success” respectively; the circles indicate prediction results; and the arrows indicate actual observed results. The 1-bit predictor toggles a prediction result when the prediction result and the real result are different. An example of state transitions is also shown in Fig. 32. Based on the prediction result, we use a speculative read scheme that is different from a general conservative read scheme. In the conservative scheme, whether a read transaction is issued or not is determined only after checking the result of the corresponding sync_read. In contrast, in the speculative read scheme, a read transaction can be issued at the same cycle as the corresponding sync_read if the prediction result is `S'. (write operations use a conservative scheme – i.e., a sync_write transaction is always initiated after the corresponding write transaction is completed.)

In the speculative read scheme, a second mechanism to handle wrong predictions is required. There are two cases of misprediction – the first case is that a prediction is 'F', but a real result is 'S'. The second case is that a prediction is 'S', but a real result is 'F'. In the first case, subsequent operations are same as in the conservative read scheme - i.e., a comm_read transaction is issued only after seeing the result of the sync_read transaction. Our second mechanism does nothing in this first case. In the second case, when a prediction turns out to be 'S', a comm_read transaction is issued in the next cycle. And then, a real result comes out as 'F'. Our second mechanism disregards the results of the previous sync_read and comm_read, and issues a new sync_read transaction. By this way, the functional correctness of our second mapping method is preserved.

This speculative read scheme, however, does not always work well. If the mis-prediction rate is too high, useless read transactions can waste memory bandwidth and



(a) Sequential buffer mapping **(b) Balanced buffer mapping at edge-level (even1)**

(c) Balanced buffer mapping at comm/sync-level (even2)

Figure 33. Three buffer mapping methods.

this overhead may overshadow the benefits provided by the parallelism. Thus, we should determine whether or not the comm/sync-level parallelism is used based on the prediction accuracy, which can be estimated at a compile-time.

To show the difference of our proposed methods compared to the conventional sequential mapping, two types of balanced mappings are illustrated in Figs. 33(b) and (c). Figure 33(b) shows a balanced mapping using the edge-level parallelism (even1), and Fig. 33(c) shows a balanced mapping using the comm/sync-level parallelism on the top of the edge-level parallelism (even2). These buffer mapping examples pertain to the IPC graph of Fig. 29. Here, *ei*,*c* and *ei*,*s* represent the communication transaction buffer and the synchronization transaction buffer, respectively, for an IPC edge *ei*.

To provide scalability, we use a modulo-operation to perform the mapping of buffers to banks. If an application requires a number of buffers that exceed the maximum number of configurable DRAM banks, then buffer mapping is done based on a modulo-*B* operation, where *B* is the total number of available banks. In this section, we only consider the temporal relationship within *B* edges - e.g., if *B* is 16, we consider the

temporal relationship of IPC edges 0 to 15 and the temporal relationship of IPC edges 16 to 31. But we do not consider the temporal relationship between the group-1 (IPC edges 0 to 15) and the group-2 (IPC edges 16 to 31). To consider both intra-group and inter-group temporal relationships is a good direction for the future work.

Any performance improvements from our proposed buffer mapping policy over the sequential mapping come at the cost of additional banks and bank under-utilization. This kind of cost/benefit analysis is useful to perform in conjunction with rapid prototyping and SoC-based design space exploration.

3.4 Evaluations

3.4.1 Simulator

Our simulator for these experiments is a time-driven simulator developed in C language. It is composed of a processor simulator, a bus arbiter, and a DRAM simulator.

A processor-side simulator is developed at a high level of abstraction; only the estimated execution times of actors are taken into account for the processor-side simulator. These estimates may be constant values or they may be drawn from probability distributions (e.g., to model the effects of infrequent events such as cache misses). Such a high-level simulation approach based on actor execution time estimates is useful in rapid prototyping for signal processing applications because it allows for accelerated processor-side simulation, and because execution time behavior in such applications has relatively high predictability. A global timer exists in the simulator, and each processor has its own local timer. The global timer is incremented by one on every cycle, while the local timer is incremented by the execution time of an actor at the end of the actor execution. To determine whether a processor advances at a given point, the simulator compares the global timer value with the processor's local timer value.

A DRAM-side simulator is the most complicated part in the whole simulator. DRAM-side simulations are carried out based on DRAMsim [55], which is a cycle-

accurate, highly-configurable, and C-based main memory system simulator. DRAMsim includes functions of the memory controller and main memory of Fig. 27. The function correctness and cycle accuracy of DRAMsim have been validated by several previous works [56, 57, 58]. In addition, interfacing DRAMsim with cycle-accurate processor simulators such as SimpleScalar [59] and GEMS [60] was also done in [61]. In this thesis, we interface DRAMsim with our processor-side simulator in the same way as in the previous work.

A priority-based arbitration algorithm without interruption is selected as the bus arbitration protocol, where priorities are determined based on initiation times of bus requests – the processor with the earliest initiation time across a set of conflicting bus requests has the highest priority. After a processor obtains mastery of the bus, it maintains mastery until finishing its command or data transfers through the bus, regardless of other pending or arriving requests from other processors. In other words, scheduling of bus mastery is non-preemptive. A block diagram of an overall system with an arbiter and detailed connections, which are implemented in our simulator, are given in Fig. 34. In our system, the address, write-data, and read-data buses are separated as shown in Fig. 34.

Figures 35(a) and (b) depicts timing diagrams, which are extracted from our simulator for the example of Fig. 29 IPC graph, in a point-to-point-connected system and in a shared-bus-based system. The timing diagram of the shared-bus-based system includes the effect of bus arbitrations. We assume that execution times of actors are 10 cycles for actor 0, 12 for actor 2, 13 for actor 3, and 5 for all communication actors (all *S** and *R**s). As shown in Fig. 35(a), multiple address and data requests can be served in the overlapped time slots in point-to-point-connected systems. But those causes bus conflicts in shared-bus systems. The bus arbiter resolves the bus conflicts as shown in Fig. 35(b). The timing diagrams are captured on the bus between processors and a memory controller. After a command is issued from a processor to a memory controller,

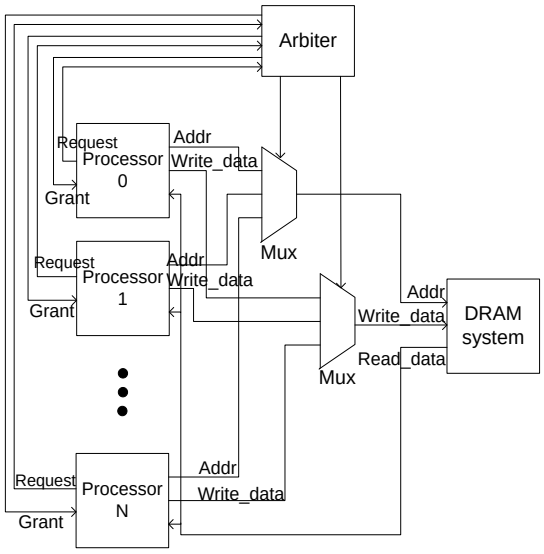
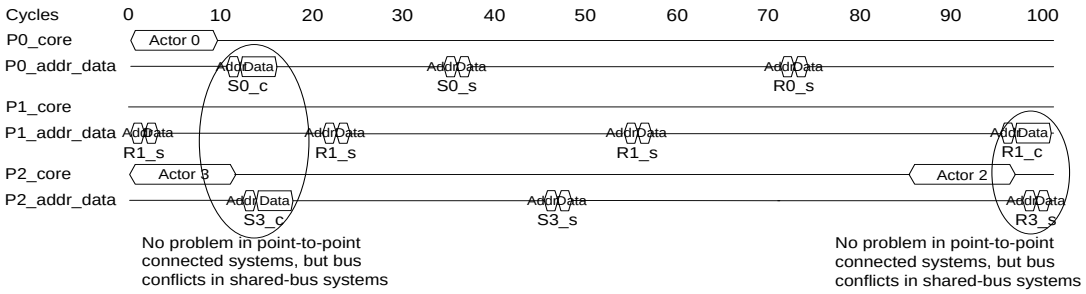
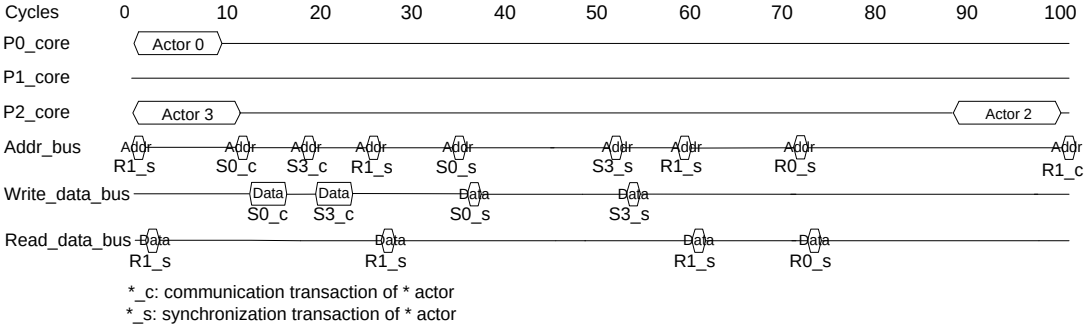


Figure 34. Block diagram of a whole system with a bus arbiter.



(a) Timing diagram in a point-to-point-connected system



(b) Timing diagram in a shared-bus-based system

Figure 35. Timing diagrams extracted from our simulator.

it takes several cycles to get a final response from a DRAM main memory - e.g., in Fig. 35(a), after S0_c is issued through the P0_addr_data line, it takes

almost 16 cycles to get a final response from a DRAM main memory. Then S0_s is issued at cycle 33.

Although timing diagrams show signal values only up to 100 simulation cycles, we always get a whole simulation log and check the validity line-by-line.

3.4.2 Simulation results and analysis

In this section, we measure application throughput for the sequential, even1 (balanced, edge-level), and even2 (balanced, comm/sync-level) buffer mapping strategies on a set of benchmarks. In our analysis, throughput is defined as the number of completed application iterations per processor clock cycle, where one application iteration corresponds to the completion of one execution of every actor. This is a common definition of throughput for HSDF graphs, since HSDF graphs typically execute iteratively across successive samples of the input signals that they are designed to process, and in HSDF, each actor executes exactly once (has a repetitions vector component equal to one) per graph iteration.

We examine the synthetic and real benchmarks shown in Table 3. We use the TGFF algorithm to generate the synthetic benchmarks [62]. The benchmark application graphs are fairly complicated with 28-68 nodes, and the numbers of processors involved during scheduling ranges from two to eight. The examples fft1 and fft2 result from two

Table 3. Benchmarks.

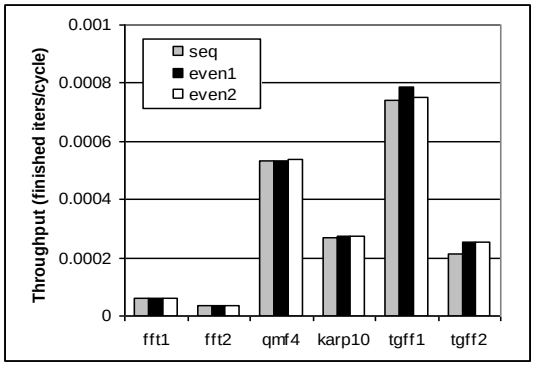
Normal	(|V|, |E|)	# of proc.	# of IPC edges	Mem-intensive	# of proc.	# of IPC edges
fft1	(28, 32)	2	16	fft1_m	4	22
fft2	(28, 32)	3	20	fft2_m	6	23
qmf4	(14, 21)	2	10	qmf4_m	4	13
karp10	(21, 29)	3	16	karp10_m	6	20
tgff1	(20, 30)	3	16	tgff1_m	6	16
tgff2	(68, 119)	4	82	tgff2_m	8	86

representative schedules for fast Fourier transforms based on examples given in [63]; qmf4 is a four-channel multi-resolution QMF filter bank [64] for signal compression; and karp10 is a music synthesis application based on the Karplus Strong algorithm [65] in 10 voices. In addition to experimenting with these "normal" benchmarks, we derive a set of "memory-intensive" benchmarks from the normal ones by doubling the number of processors targeted in the scheduling stage.

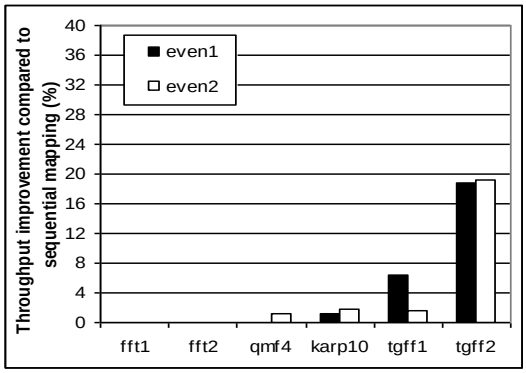
DRAMsim parameters are set as follows: DRAM type = SDRAM, DRAM freq. = 100 MHz, # of ranks = adjusted, # of banks = 4, # of rows = 8192, # of columns = 512, transaction scheduling policy = first-come first-served, row buffer management policy = close page, address mapping policy = SDRAM base map, and refresh policy = all ranks and all banks at a time. All simulations are done for 1,000,000 simulation cycles.

Figure 36(a) shows the measured throughput for the three different buffer mapping policies on the six normal benchmarks. Figure 36(b) shows percentage improvements of the even1 and even2 mappings compared to the sequential mapping. As shown in Fig. 36(b), the percentage improvement is generally very different from application to application; about 15% in tgff2 but 0% in fft1 and fft2. This is largely affected by the total execution time of all actors on a given processor.

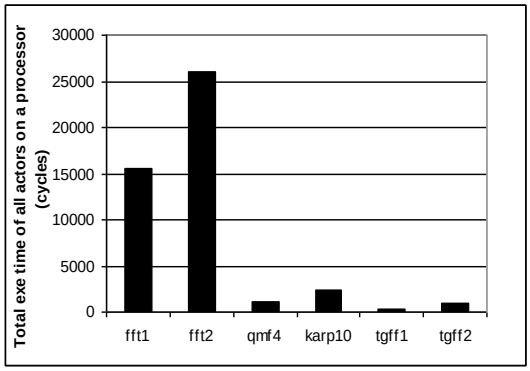
Figure 36(c) shows the average total execution time of all actors on a processor. For example, even if we save 25,000 cycles by applying the even1 mapping, this value is roughly translated to just one less iteration in fft2, but about 25 less iterations in tgff2. So the large execution times of fft1 and fft2 result in almost 0% improvements of the even mapping policies compared to the sequential mapping. The other interesting point is that the even2 throughput is smaller than the even1 throughput in tgff1. This is due to the relatively low prediction accuracy of sync_read checks; the prediction accuracy in tgff1 is about 83%, but for the other benchmarks, the accuracy is almost 90%. As described in the



(a) Throughput



(b) % improvement

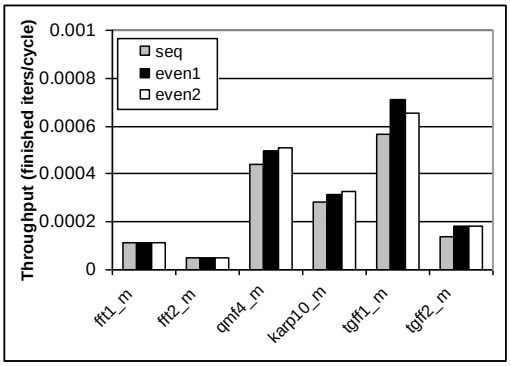


(c) Total execution time of actors

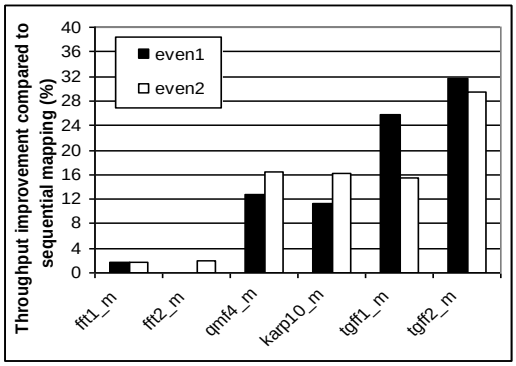
Figure 36. Simulation results for normal benchmarks.

previous section, a low prediction accuracy can cause useless read transactions to waste memory bandwidth significantly.

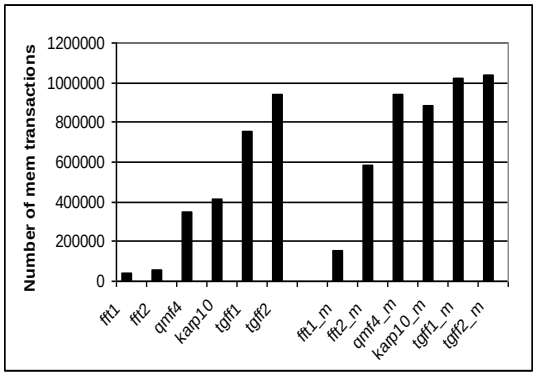
To see the effect of applications' memory intensity on performance, we perform the same simulations on the six memory-intensive benchmarks. Figure 37(c) simply shows the increased memory intensity of the memory-intensive benchmarks compared to the corresponding normal benchmarks. The measured throughput is shown in Figs. 37(a) and (b). When examining these results, we see that first of all, percentage improvements in the memory-intensive benchmarks are larger than those in the corresponding normal benchmarks except for fft1_m. In fft1_m, the even1 mapping policy does not show any improvement mainly because of its large total execution time. In tgff1_m, the even2



(a) Throughput



(b) % improvement



(c) Total # of memory transactions at sequential mapping

Figure 37. Simulation results for memory-intensive benchmarks.

throughput is smaller than the even1 throughput due to the relatively high misprediction rate of the predictor.

Overall, the simulation results show that the proposed buffer mapping policy is very useful, especially in memory-intensive applications with relatively small total execution time of actors. Whether or not the even2 mapping is used should be determined depending on sync read prediction accuracy.

3.5 Related work

Buffer mapping has a major impact on system performance. To understand this impact, it is useful to view a DRAM main memory system as a finite state machine whose next state is determined by the current state and the incoming memory operation command.

One factor that complicates memory analysis is that the DRAM main memory exhibits non-uniform access latency depending on access history.

Various work has considered SDF buffer management, such as those reported in [66, 67, 68]. In recent years, there has also been significant emphasis on systematically exploring trade-offs between throughput and buffer memory requirements in SDF graphs (e.g., see [69, 70, 71]). More specifically, previous work on SDF techniques computes or measures application throughput assuming zero or uniform access latency to the DRAM main memory system [50, 54]. Furthermore, the impact of different IPC buffer mapping policies on performance has not been considered much. In general-purpose computing, most of the research on improving memory performance has focused on memory controller techniques, such as command scheduling and memory address interleaving [12, 13, 14]. In addition, there is no previous work on SDF techniques to lower energy consumption related to the buffer mapping method.

Contributions of this thesis are as follows:

1. We consider practical aspects of DRAM main memory systems for measuring application throughput in multiprocessor DSP systems.

2. We propose high-throughput and low-energy buffer mapping methods considering the practical aspects of DRAM main memory systems.

3.6 Summary

We have proposed the high-performance buffer mapping policy for SDF-based DSP applications that are targeted to multiprocessor systems supporting the shared-memory programming model. The proposed policy exploits bank and rank concurrency of contemporary DRAM main memory systems according to the careful memory system modeling and parallelism analysis. We use a graph coloring technique to analyze the IPC (inter-processor communication) edge-level parallelism and a 1-bit predictor to analyze the comm/sync-level parallelism.

In our experiments, we measured application throughput on both synthetic and real benchmarks. The simulation results show that the proposed buffer mapping policy is very useful in terms of throughput, especially in memory-intensive applications with relatively small total execution time of actors. Whether or not the even2 (comm/sync-level) mapping approach is used should be determined based on sync_read prediction accuracy, which can be estimated at a compile-time.

The performance improvements of the proposed buffer mapping policy are achieved in general at the cost of additional banks and bank under-utilization.

CHAPTER IV

INTERCONNECTION NETWORKS

Our goal in this section is to design a high-performance Turbo decoder and FFT processor. Those two applications are very popular embedded applications. To achieve the goal, we consider a network-centric approach to efficiently connect multi- or many processing elements.

We first design a crossbar-based parallel Turbo decoder and then design a mesh-based parallel FFT processor.

4.1 Network-centric parallel Turbo decoder

In this section, we propose a parallel Turbo decoder architecture based on an interconnection network, which can accommodate arbitrary interleaving/deinterleaving schemes during run-time.

Interleaving/deinterleaving of extrinsic values in Turbo decoding can lead to access conflicts to network resources. We consider a virtual output queuing (VOQ) input-queued crossbar switch as an interconnection network to handle the conflicts. We use the parallel iterative matching (PIM) algorithm to configure the crossbar switch.

4.1.1 Overall architecture

Even though a single-core Turbo decoder can satisfy increasing throughput requirements with a very deep pipelining, it has the power wall and development/verification time problems. A multi-core Turbo decoder with relatively simple component cores can be an alternative. In this section, we consider a crossbar switch-based multi-core Turbo decoder.

In our network-centric parallel Turbo decoder, MAP cores are placed at one side, and shared memories are placed at the other side as shown in Fig. 38. The number of

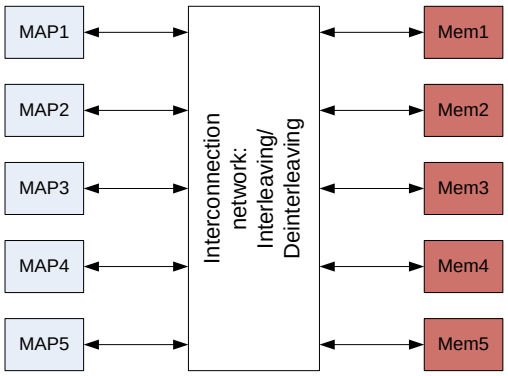


Figure 38. Dancehall configuration of a parallel Turbo decoder.

memories is chosen as equal to the number of MAP cores according to the analysis in the previous work. We call this a dancehall configuration. Figure 38 shows an example with five MAP cores and five memory banks. Our architecture is based on a single soft-input soft-output (SISO) scheme, i.e., a single MAP core fulfills computations for both component decoders in a time-multiplexed way.

We exploit the sub-frame-level parallelization that was explained in Section 2.2.2. Each MAP core computes alpha, beta, gamma, and extrinsic values corresponding to each sub-frame. And it sends out extrinsic values to the interconnection network, which are used as a priori information to the other component decoder for iterative decoding. The extrinsic values are interleaved by writing them in a permuted order to the memory, and reading them in a sequential order from the memory. The cross-iteration reference method [72] is used to initialize the alpha and beta metrics at the boundary between sub-frames.

The memory banks are used for storing extrinsic values. The size of each memory bank is the same. Other kinds of memories in a Turbo decoder such as input buffers and alpha (or beta) memory are included inside MAP cores.

The most critical issue in the dancehall configuration is how to handle communication traffic on the interconnection network. Several extrinsic values (e.g., five

in Fig. 38) simultaneously arrive at input ports of the interconnection network every cycle during backward recursions. This simultaneous arrival of the extrinsic values can cause conflicts of network resource usage. The interconnection network should handle conflicts and forward the extrinsic values to the right memory banks according to an interleaving rule. The design of a fully-adaptive interconnect network is the theme of this section. A fully-adaptive network dynamically accommodates arbitrary interleaving schemes during run-time. We will describe the interconnection network architecture in more detail in the next section.

4.1.2 Interconnection network architecture

The interconnection network in our parallel Turbo decoder is responsible for interleaving/deinterleaving of extrinsic values. Henceforth, we only consider interleaving since deinterleaving is implemented in the same way as interleaving. The interleaving can generate both internal and external conflicts of network resources. Internal conflicts occur inside switches and external conflicts occur at network output ports, i.e., the memory input ports in Fig. 38. If, for example, routing of several extrinsic values needs the same network resources (such as switches) at the same time, *internal conflicts* occur. If destined memory bank IDs of several extrinsic values are identical at the same cycle, *external conflicts* occur.

To resolve internal conflicts, we choose a proper type of switching fabric among non-blocking networks. Any unconnected input can be connected to any unconnected output without affecting the existing connections in a *strictly non-blocking* network or by rearranging the existing connections in a *rearrangeably non-blocking* network. We select one of the strictly non-blocking networks - crossbar switch - as the interconnection network in our proposed parallel Turbo decoder. We assume that a crossbar switch is

readily implemented with 150 ports or less that is equal to the number of MAP cores to be considered in this section.

External conflicts can be resolved by two different approaches: by a conflict-free interleaver or by a buffer-based scheme. First, we can avoid external conflicts from the beginning by designing a conflict-free interleaver [73, 74, 75] in which all destined memory bank IDs of extrinsic values at the same cycle are totally different. This static method has advantages of higher throughput than does a buffer-based scheme. But this method is too restrictive to be used, i.e., a special interleaver must be redesigned every time any related parameter is changed. Second, we can resolve occurred external conflicts during run-time by buffering unserved extrinsic values [76, 77]. Since our goal is to achieve a fully dynamic solution, we select the buffer-based scheme. Among three ways to implement buffers: input-queued, output-queued, and combined-input-output-queued (CIOQ) schemes, we consider an input-queued scheme and use a FIFO queue as a buffer. In the input-queued scheme, every input has its own FIFO queue. Unserved extrinsic values by a network at each input are stored in its FIFO queue and tried to be retransmitted in the future cycles. Since the only head extrinsic value in each FIFO queue is eligible for the transmission, the input-queued scheme suffers from the head-of-line (HOL) blocking problem, which results in poor throughput [78].

To resolve the HOL blocking problem, we use the virtual output queueing (VOQ) [79]. In the VOQ scheme, each input port has multiple FIFO queues. Each FIFO queue stores extrinsic values destined to one of the output ports. As a result, the HOL problem is eliminated – extrinsic values destined to free memory banks are no longer blocked by other extrinsic values destined to occupied memory banks.

In summary, the VOQ input-queued switch with a crossbar fabric is used as the interconnection network of the proposed Turbo decoder as shown in Fig. 39, where Q(*i*,

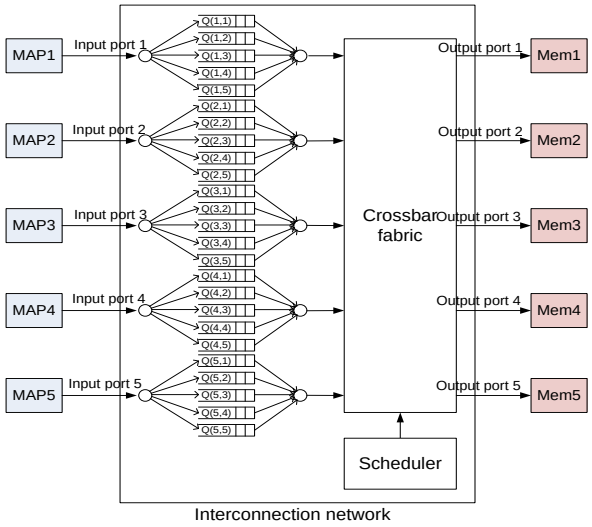


Figure 39. Proposed architecture of a parallel Turbo decoder.

j) is the *j*th queue (destined to the *j*th output port) of the *i*th input port. The interconnection network is mainly composed of input queues, a crossbar switch, and a scheduler.

4.1.3 Scheduling algorithm - PIM

After determining hardware configurations, we should consider scheduling, switching, and routing algorithms. Since we use a crossbar switch as an interconnection network, a scheduling algorithm needs to be considered.

A scheduling problem through the crossbar switch can be seen as a matching problem in a bipartite graph. A request graph can be represented as a bipartite graph as

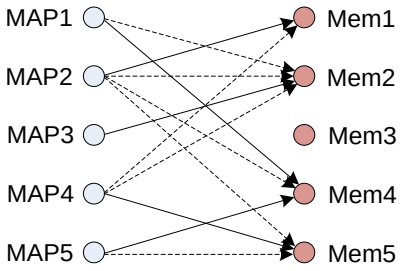


Figure 40. Bipartite graph of requests.

shown in Fig. 40 for the example of Fig. 39. The nodes at the left-hand side indicate MAP cores and the nodes at the right-hand side indicate memory banks. The solid lines represent new requests that are generated at the current cycle and the dashed lines represent old requests that were not served at the previous cycles. The number of dashed lines is various, but the number of solid lines is always equal to the number of MAP cores – every MAP core generates a single new extrinsic value every cycle during backward recursions.

In this thesis, we use the parallel iterative matching (PIM) algorithm, one of the maximal matching algorithms, because it is feasible to implement and it is the basis of other iterative matching algorithms. For a detailed description of the PIM algorithm, we refer readers to [80]. Although maximum matching algorithms such as the Maximum size and the Maximum weight generally show better performance than maximal algorithms, they are not considered here because of their *running time complexity* $O(N^{2.5})$ for the Maximum size [81] and $O(N^3 \log_2 N)$ for the Maximum weight [82], where N is the switch size.

Although the PIM algorithm is applicable to Turbo decoding from the viewpoints of implementation feasibility, we must also consider stability, starvation, and fairness issues. The mathematical analysis of these issues is out of our scope. Instead, we check these issues in Turbo decoding based on experimental results shown in Section 4.1.4.

The last thing to consider in the PIM algorithm is the number of iterations (the definition of the iteration is different from that of the Turbo decoding iteration). We can either execute a fixed number of iterations during a whole decoding process or execute iterations until a maximal matching is found. The latter method results in various run-time from one set of extrinsic values to another set. We use a more predictive one, the first method.

4.1.4 Evaluations

In this section, we measure throughput [in bits/cycle] of the proposed Turbo decoder with the PIM algorithm under 3GPP-style block interleaving.

We develop a cycle-accurate simulator in C language based on the existing functional simulator of Turbo decoder that was used in [84]. In our simulations, execution time is composed of computation time spent by the MAP cores and communication time spent by the interconnection network. We assume that memory access time is uniform. Simulation parameters are set as follows:

Encoder: two identical RSCs, generator = [1, 11/13], constraint length = 4, code rate = 1/3,

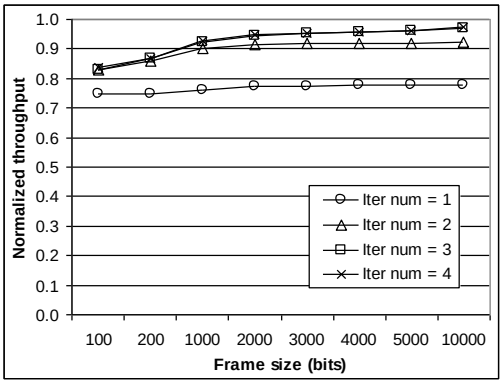
Decoder: Log-MAP algorithm, number of iterations = 8,

Channel: Gaussian channel with E_b/N_o = 0.5dB.

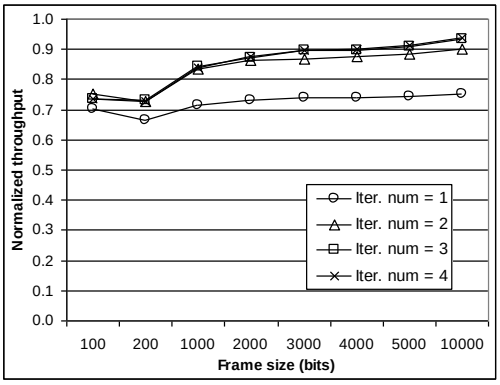
We measure throughput over various frame size = 100 ~ 10,000 bits and various number of MAP cores = 5 ~ 150.

The first set of experiments are done to determine the number of iterations in the PIM algorithm required to find a good matching (it can be maximal or not). Figure 41 shows measured throughput of the proposed parallel Turbo decoder with a different number of PIM iterations at 10, 50, and 100 MAP cores. The measured throughput is normalized by the ideal-case throughput (maximum attainable throughput) - in the ideal case, all newly generated extrinsic values every cycle are served instantly. As shown in Fig. 41, throughput is saturated after three iterations over all frame size regardless of the number of MAP cores. This performance pattern is almost consistent with results in [85]. According to our simulation results, we set the number of iterations in PIM as three in the following simulations.

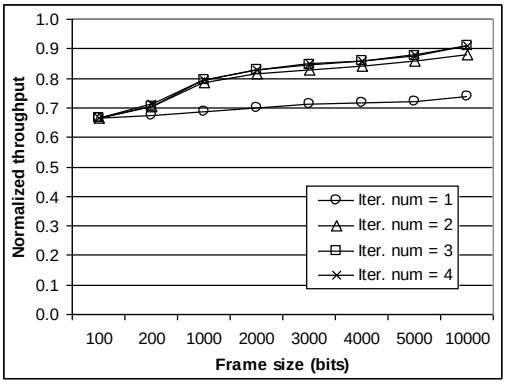
The next set of simulations is to show scalability of our architecture in terms of throughput – the effect of increased number of MAP cores on throughput. Figure 42(a)



(a) Number of MAP cores = 10



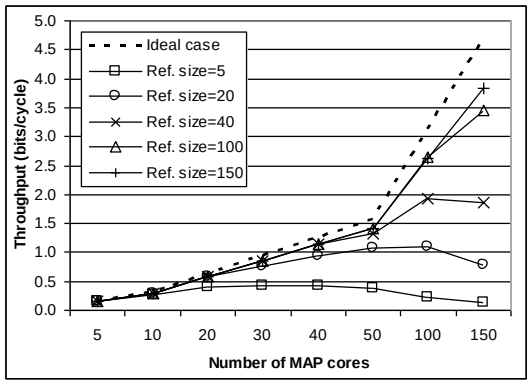
(b) Number of MAP cores = 50



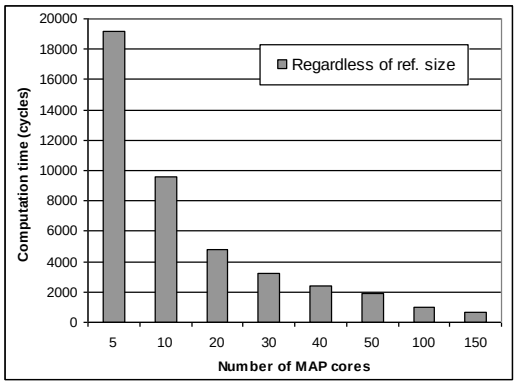
(c) Number of MAP cores = 100

Figure 41. Normalized throughput.

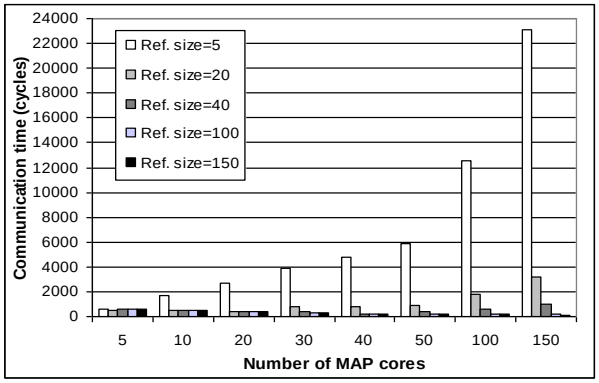
shows measured throughput [in bits/cycle] at five different reference switch size. As shown in the figure, the increasing number of MAP cores does not always guarantee throughput improvements. This is due to the fact that the negative effect of increased communication time may overshadow the positive effect of decreased computation time on throughput in some cases as shown in Fig. 42(b) and (c). The turning point is different according to the reference switch size. For example, throughput starts decreasing from 40 MAP cores at reference switch size = 5 and from 100 MAP cores at reference switch size = 20. Therefore, we must carefully determine the right number of MAP cores in our proposed architecture according to the reference switch size, i.e., feasible scheduler speed. For example, if the current technology can accommodate three iterations in one



(a) Throughput



(b) Computation time



(c) Communication time

Figure 42. Scalability tests in terms of throughput – frame size = 3000bits, # of iterations in PIM = 3.

cycle at 40 x 40 or less crossbar switch, using more than 100 MAP cores is meaningless. This scalability analysis is equally applied to other cases with different frame size.

If we can estimate the operating clock frequency of our parallel Turbo decoder, we can calculate absolute throughput (in bits/sec) from the results in Fig. 42(a). The previous work [86] presents a very similar parallel architecture to ours; main difference is that their architecture uses one more crossbar switch and our architecture uses the VOQ input queue. Therefore, we can roughly estimate our clock frequency as 200 MHz at 32 MAP cores based on their synthesis results with a 65 nm technology. In the worst case – the case with reference switch size = 5, our proposed architecture serves the absolute

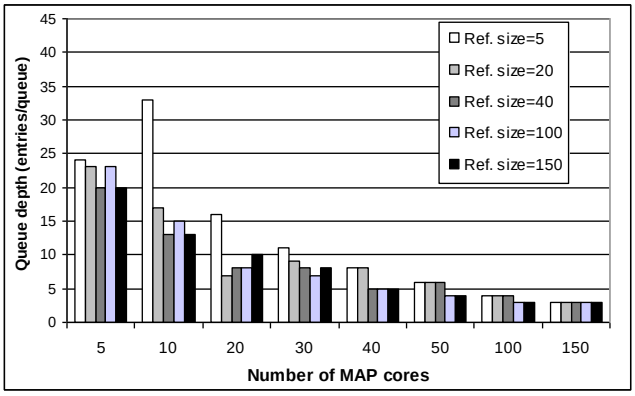


Figure 43. Maximum component queue depth.

throughput ~= 0.42 bits/cycle x 200 MHz = 84 Mbits/sec with 32 MAP cores and eight Turbo iterations, which is acceptable considering current applications' requirement and the other previous work [76, 87, 88, 89]. Although this estimation of clock frequency is very rough, it can show the range of attainable throughput by our proposed architecture at least.

Since the increasing number of MAP cores results in the dramatic increase in communication time, especially at the lower reference switch size as shown in Fig. 42(c), the input queue size could be obstacle to implementation. We measure the maximum, component queue depth (in number of entries per queue) as shown in Fig. 43. We use a circular buffer as a component input queue, where head and tail pointers wrap around according to modular operations. The simulation result shows that the component queue depth decreases as the number of MAP cores increases, which is opposite to our prospect. This is due to the fact that the positive effect of traffic distribution is bigger than the negative effect of communication time increase on the component queue depth at the large number of MAP cores. The input queue size is not an implementation obstacle.

4.1.5 Related work

Our proposed Turbo decoder resolves network resource conflicts under arbitrary interleaving schemes during run-time and we consider ASIC implementations of the

proposed Turbo decoder. There is several previous work related to this dynamic solution. Neeb et al. [76] considered strictly orthogonal networks such as mesh, torus, and cube networks as interconnection networks of parallel Turbo decoders. They used an input- or output-queued crossbar switch as a component switch with the SLIP scheduling algorithm [83]. They mainly focused on performance comparisons of different routing algorithms. Moussa et al. [77] used the butterfly and the modified Benes networks as an interconnection network of their parallel turbo decoder. To handle external conflicts, they used FIFO buffers in the butterfly network and off-line scheduling (semi-dynamic, between a conflict-free interleaver design and a buffered method in terms of dynamic level) in the Benes 2*N*-*N* network. The other work [86, 87] also considers ASIC implementations, but not a dynamic solution, i.e., they are designed based on conflict-free interleavers. The other previous work [88, 89] considered software (programmable) implementations of parallel Turbo decoders on VLIW and SIMD machines.

4.1.6 Summary

In this section, we proposed the parallel Turbo decoder architecture based on the VOQ input-queued crossbar switch, which can accommodate arbitrary interleaving/deinterleaving schemes during run-time. The proposed architecture dynamically resolves both internal and external conflicts of network resources. And we found that the PIM algorithm can be applied to Turbo decoding.

We measured throughput of the proposed Turbo decoder with the PIM algorithm under 3GPP-style block interleaving by using our cycle-accurate simulator that makes fast and wide design space exploration possible. Simulation results demonstrate that the feasible scheduler speed is a very critical parameter when determining the right number of MAP cores in the parallel architecture. And the input queue size is not an implementation obstacle, even at the large number of MAP cores.

4.2 Network-centric FFT processor – electrical mesh network

In Section 4.1, we proved only small number of MAP cores are needed in Turbo decoding to meet throughput requirements of all contemporary communication standards. In other words, scalability is not an issue, so we used the crossbar-based indirect topology in designing a high-performance network-centric Turbo decoder.

In this and the next sections (Sections 4.2 and 4.3), we consider another popular and core application in the embedded computing world – Fast Fourier Transform (FFT). A parallelization degree of FFT is much bigger than that of Turbo decoding, so more number of processing elements can boost the performance. As a result, a good scalability is essential in designing high-performance FFT processors. To handle a scalability issue, we apply another topology – mesh. A mesh network in this section is a general electrical network system. (On the contrary, a mesh network in Section 4.3 is an optical network system.)

The following subsections explain three major parameters – a topology, a routing, and a flow control - to characterize our interconnection network.

4.2.1 Topology

We consider a 2D mesh that is widely used in existing computing systems. A *k*-ary 2D mesh network is composed of k^2 nodes in a regular two-dimensional grid with *k* nodes in each dimension and nearest neighbors are connected by a channel as shown in Fig. 44. Every black rectangular node indicates a switch, every white rectangular node indicates a processing element, and every gray rectangular node indicates an off-chip DRAM system.

Mesh networks provide bidirectional channels and a good path diversity. The latency and throughput are very good for local communication traffic (i.e., between neighbors), but bad for remote traffic [90]. Applying mesh topology has several implications. In a layout perspective, a general CMP style is well matched to the mesh topology. Another implication is that routing and flow control algorithms are more complex than those in indirect topologies, so in addition to a processing core and a memory, a router is another major component in mesh-based NoCs.

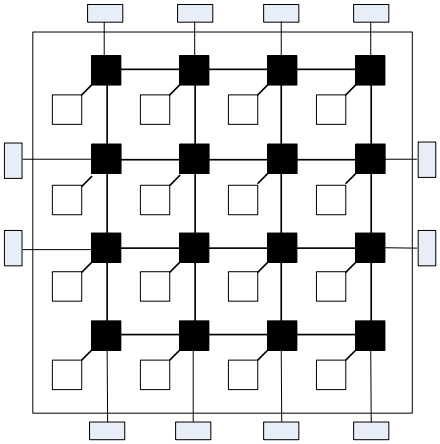


Figure 44. 2D Mesh topology.

4.2.2 Routing and flow control algorithms

A routing algorithm is one of the main factors to determine how close we get to the upper bound performance that is determined by a topology. A routing algorithm determines an optimal path from a source node to a destination node. The meaning of "optimality" is different depending on scope – in local scope, it means a shorter path length, but in global scope, it is more related to good load balancing. Therefore good routing algorithms should consider both by a trade-off analysis. A flow control algorithm is one that prevents a fast sender outrunning a slower receiver. It manages the allocation of network resources - such as buffers and channels – to a packet. Analyzing the impact of various routing and flow control algorithms on performance is out of our scope. In this section, we simply use dimension-order routing and virtual-channel flow control algorithms.

The dimension-order routing algorithm is one of the deterministic routing algorithms that always select the same path from a particular source to a particular destination even though there are multiple possible paths. Deterministic routing algorithms are very common in practice because of easy implementation, despite of poor performance. In the dimension-order routing algorithm for a 2D mesh topology, a routing path is selected dimension-wise – a packet is routed in one dimension (along the X-axis or Y-axis) until

reaching to a proper coordinate, and then routed in another dimension reaching to the final destination (along the Y-axis or X-axis).

The virtual-channel flow control [91] is one of the buffered flow control algorithms. By providing multiple buffers (called virtual channels) per each physical channel, it allows packets to pass other blocked packets.

4.2.3 Circuit switching vs. packet switching schemes

A traditional circuit-switched network was first introduced in telephone systems for transferring voice data. In a circuit-switched network, a whole path is established from a source to the final destination and resources are allocated to the connection before sending data. The resources remain allocated until the data transfer completes. Then the path is finally released (i.e., torn down). In circuit switching, data buffers at intermediate routers are not required, which results in some improvements in silicon area and power consumption. (Only small-size buffers are needed for path-setup request and acknowledge tokens.) In addition, all data of the same message follow the same path and they arrive at the destination in order, so additional overheads (such as data reordering at the destination node) are not needed. One of the big disadvantages of the circuit switching is that resources are tied up to one pair of source and destination during a whole session even though there is no data flow intermediately. So other connection requests are blocked and resources are wasted. Another disadvantage is that complex signaling is required to establish and maintain the end-to-end path. For the circuit-switching scheme to be effective, a message size should be enough large to compensate the path setup and teardown overheads.

Packet switching can overcome those disadvantages of circuit switching. Principles behind packet switching at a chip level originate from computer networks, so chip-level packet switching basically follows the OSI seven-layer model [92]. In packet switching, a message is segmented into smaller units called packets regardless of the original data

type (e.g., the type can be voice or digital data). There are mainly two different types of packet switching schemes – connection-oriented or connectionless.

Datagram packet switching is also known as connectionless switching. In contrast to circuit switching, the end-to-end connection is not needed. In the datagram switching scheme, each packet includes a destination address in its header. Each switch should map the destination address to the outgoing port for every incoming packet. A routing table in each switch includes this mapping information. If network resources – such as downstream buffers located at the next router – are unavailable, the packet should wait in the current buffer. A packet can be sent whenever the next outgoing port is available. In contrast to circuit switching, network resources are only allocated when there is a packet to send. So resource usage is more efficient than in circuit switching. Furthermore, complex signaling to maintain the end-to-end path is not required any more. However, datagram switching has some disadvantages. Each packet of a message is routed independently, i.e., each packet even of the same message can follow different paths, so those packets can arrive at the destination out of order. In addition, data buffers are required at every router, which is adversary to a general SoC design rule – silicon area for communication components should be minimized. Another disadvantage of datagram switching is that the header of every incoming packet should be examined to determine the next outgoing port at a line rate. Examples of commercial datagram packet-switched systems are Internet Protocol (IP), Ethernet, and User Datagram Protocol (UDP).

Another sub-scheme of packet switching is virtual-circuit (VC) packet switching that is also known as connection-oriented packet switching. It uses the similar concept of circuit switching to deliver packet data. Virtual-circuit switching is similar to traditional circuit switching in the sense that a whole path (called virtual circuit) is set up before sending the first packet. On the contrary to circuit switching, resources are not allocated

physically while a message can make a reservation of the resources. As a result, virtual-circuit switching provides efficient resource usage as in datagram packet switching. But the expense is that some packets may wait in buffers for congested resources to be freed. Each packet carries a virtual circuit identifier instead of a whole destination address, so routing table area cost is reduced. The circuit identifier is changed by each switch during data forwarding. Since an end-to-end VC establishment and tear-down are required, a little more complex signaling protocol is needed. X.25 and Asynchronous Transfer Mode (ATM) are examples of the system implementing the virtual-circuit switching scheme.

In this section, we examine the traditional circuit switching and datagram packet switching schemes on the top of the electrical mesh topology.

4.2.4 Off-chip memory interface

An off-chip memory system interfaces with a NoC via a network interface (NIF) as shown in Fig. 45. Since in our system, any data transfer between a node (meaning processing element + router) and an off-chip memory goes through the NoC router (this is true even for local transfers), the memory system should support a network protocol. In our system, the memory controller is responsible for supporting the network communication protocol in addition to its normal functions – such as memory transaction ordering. The inner structure of the DRAM memory system is same as one explained in Section 3.1 and Fig. 27.

We map the FFT application onto our electrical mesh network system and measure

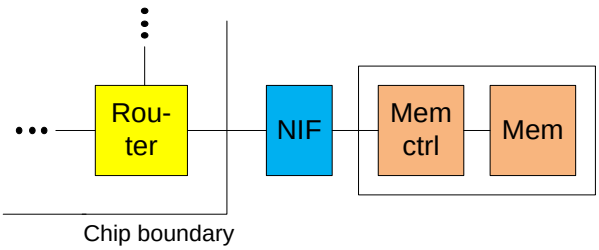


Figure 45. Off-chip memory interface with an electrical NoC.

the performance in Sections 4.3.7 and 4.3.8.

4.3 Network-centric FFT processor – hybrid mesh network

In this section, as another approach to improve performance, we apply an optical fiber as an interconnection network medium. Photonics provides several advantages compared to electrical counterparts. The power dissipated in optical networks is distance-independent. Furthermore, power consumption is also data-rate-independent and a photonic switch does on/off only once per message. Finally, photonics provides a high bandwidth density [in bps/μm].

A topology of optical network systems that we consider in this section is the mesh. And off-chip main memory systems are distributed around the NoC periphery in the same way as in the electrical network systems (Figs. 44 and 46).

4.3.1 Implications of using an optical fiber as an interconnection medium

Since processing elements and memories are still manufactured by CMOS electronics (optical processing elements and optical memories are still beyond feasible implementation), using optical interconnection networks requires the integration of CMOS electronics and optics. Advances in silicon-based nano-phonic technology make

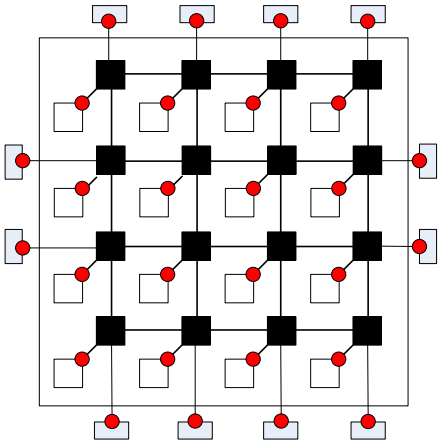


Figure 46. Photonic Mesh NoC (red circles indicate E-O and O-E conversion points).

the integration feasible. In addition, the integration implies the necessity of O-E and E-O conversions at some interfaces (indicated as red boxes in Fig. 46).

Another major component – a switch – has a different aspect; fully-optical switches have been researched in [93, 94] and fully-optical implementation of switch is more feasible than processing elements and memories. Here, "a fully-optical" implementation means that both control – e.g., routing - and data path mechanisms are optically supported. The optical control requires all control-related processing to be done optically – e.g., optical routing and optical flow control algorithms are needed. As a result, the implementation cost of a fully-optical switch is too high because of use of exotic optical materials.

An economic way to overcome the problem of fully-optical switches is to separate a control plane and a data plane. An electrical network is used in the control plane to set up and tear down a data path, but an optical network is used to carry data. This concept of electrically controlled optical switch was proposed by [95, 96] and they call it "a hybrid switch". We use this in our optical network system. By separating the two planes, we can remove optical control requirements, but still exploiting advantages of an optical medium when transferring data. However, note that requirements of optical buffering in the data path may still be an issue. By choosing an appropriate switching scheme, we can remove the optical buffering requirement as will be explained in Section 4.3.2.

In our context, the "hybrid" indicates the mixture of two different media – electric and photonic - in designing interconnection networks. Previous work also applied a hybrid concept to designing high-performance, low-energy interconnection networks, but the definition of hybrid is different from ours. For example, in Firefly [97] multiple cores are grouped into a cluster. Intra-cluster communications are handled by an electrical network, but inter-cluster communications are handled by an optical network.

4.3.2 Communication protocol in the hybrid network

The hybrid network is based on circuit switching because of the following reason. In packet switching, data is transferred in a store-and-forward manner (only after a whole packet is received, the next transfer of the packet can be started to an outgoing port), so data buffers are required. Although some previous work demonstrated an optical buffer implementation using optical fiber loops, buffer requirements make packet switching inappropriate for optical network systems. Instead a circuit-switching scheme is naturally well suited to photonic networks.

The communication protocol described in Section 4.2.3 also works in our hybrid network; In the hybrid network, path setup and tear-down are achieved by communicating a control packet electrically. While setting up the path, an electrical router sets up the corresponding optical switch (5 x 5 crossbar in the mesh network) for end-to-end data transfers. Signal transmissions in the data path are actually via optical fibers without any intermediate buffering, and E-O and O-E conversions are needed at the interface between PEs and optical switches at the beginning and end nodes, respectively.

4.3.3 Wavelength division multiplexing (WDM)

In addition to the hybrid concept, another unique feature in photonic networks is an introduction of a new multiplexing scheme. Actually, the basic idea of the scheme is not new – WDM [98] is an optical version of frequency division multiplexing (FDM). But the range of frequency is orders of magnitude higher than that in FDM - normally a radio frequency range is considered in FDM, but an IR range is considered in WDM. WDM is a technology that multiplexes several optical signals with different wavelengths into a single optical fiber. In theory, system bandwidth is linearly proportional to the number of optical signals to be transmitted at the same time. Contemporary photonic components – such as laser, modulator, and detector - support WDM. We will check this in Section 4.3.5. In this thesis, we apply WDM to our hybrid network system.

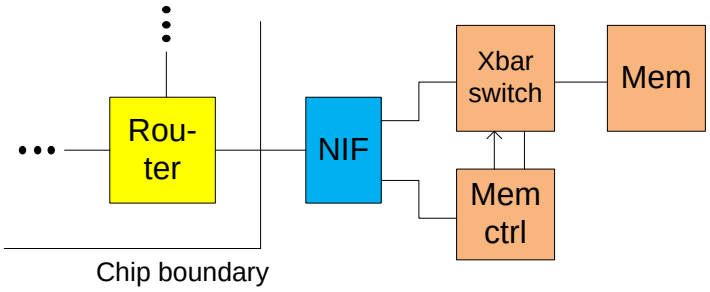


Figure 47. Off-chip memory interface with a hybrid NoC.

4.3.4 Off-chip memory interface - optically interfaced memory

The final issue that needs to be addressed is an optically interfaced memory. We mainly focus on off-chip main memory systems.

The development of photonic memories is still under way, so a general electronic DRAM is still popular candidate as an off-chip main memory in optical or hybrid network systems. Therefore, the interface between optical switches and electrical DRAMs should be designed including the function of E-O and O-E conversions. Another important point is that all DRAM memory accesses should go through a NoC, so a network protocol should be involved in DRAM memory accesses in some ways. (Even local accesses to DRAMs should go through a NoC in our hybrid network system. Some previous work tackled this issue by separating and providing different access mechanisms for local DRAM accesses and remote DRAM accesses.)

To achieve both functions - E-O and O-E conversions, and a NoC protocol support -, we use the system configuration as shown in Fig. 47. The memory controller is responsible for the NoC protocol support such as circuit-path setup and teardown. The memory module (Mem in Fig. 47) is responsible for the E-O and O-E conversions. To do so, we use a circuit-accessed memory module (CAMM) as the memory module. For further information about CAMMs, we refer reader to [99].

4.3.5 Optical building components

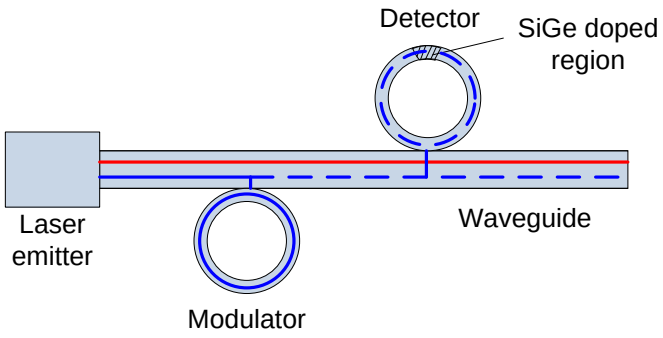


Figure 48. Basic photonic system.

In this section, we introduce photonic component devices required to build our hybrid network system. We will see what kind of characteristics at a device level is provided for supporting WDM.

The nanophotonic component devices needed to build photonic NoCs are light source, modulators, detectors, and waveguides (optical fibers) as shown in Fig. 48. Another main component is a photonic switch that is made of a set of ring resonators and waveguides.

To save manufacturing costs, developing those photonic devices in a CMOS-compatible process is essential since the CMOS process is already used in electrical component manufacturing.

A laser emitter is an obvious choice for a light source. It produces an optical carrier on which electrical data signal is modulated. The optical carrier has a specific wavelength. To exploit WDM transmissions, a laser emitter is designed to produce multiple wavelengths at a time, which are transmitted simultaneously through a single WDM waveguide. For example, in Fig. 48 two optical signals with different wavelengths are generated from the laser emitter. We assume the WDM laser emitter is located off-chip and coupled to the chip by an optical fiber [100].

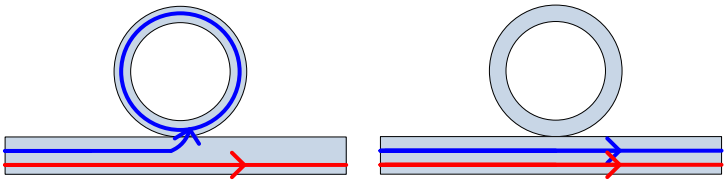
A waveguide is an optical wire that carries optical signals. To achieve good confinement and small loss of the light, a waveguide is fabricated using two different

materials: crystalline silicon (a material with a high refractive index) as a core and silicon-oxide (a material with a low refractive index) as a cladding. A single WDM waveguide can carry multiple optical signals with different wavelengths at a time. For example, in Fig. 48 the single WDM waveguide carries the two optical signals simultaneously.

Before describing a modulator, a detector, and a switch, we first explain a ring resonator that is a main building component of those three devices - a ring shape in Fig. 48. One ring resonator is shown in the modulator and the other is shown in the detector. A ring resonator has a certain resonance frequency (i.e., resonance wavelength) that is determined by a radius of the ring, a refractive index of the ring material, and a thermal tuning. Each ring resonator captures only one signal (among several signals transferred through WDM waveguides) that has a wavelength equal to the ring's resonance frequency – i.e., each ring resonator is designed to operate on only one wavelength.

A ring resonator is in either an on-resonance status or an off-resonance status controlled by injecting charges into the ring – i.e., the ring's resonance frequency can be shifted a little bit by injecting electrical current. In the on-resonance status, one of the optical signals (the blue signal in Fig. 49(a)), which has a wavelength equal to the resonance wavelength, is coupled into the ring and the optical signal in the ring eventually vanishes. In Fig. 49, the two wavelengths are shown to indicate WDM transmissions. The blue signal is the ring's interest – i.e., the signal with the same wavelength as the ring's resonance frequency. In the off-resonance status, the resonance frequency of the ring is slightly shifted, so the optical signal, which was coupled into the ring during the on-resonance status, is now passed through the WDM waveguide as shown in Fig. 49(b).

A modulator accomplishes E-O conversions – it modulates (or encodes) electrical information onto an optical signal (carrier) with a specific wavelength. For WDM transmissions, a wavelength-selective modulator based on a ring resonator is needed. In



(a) On-resonance status **(b) Off-resonance status**

Figure 49. Ring resonator.

this type of modulator, a ring resonator is coupled with a waveguide as shown in Fig. 48. By switching between the on-resonance and off-resonance status, the modulation of a specific wavelength can be accomplished.

A detector accomplishes O-E conversions. A ring resonator is also used to build a photo detector. For WDM transmissions, a wavelength-selective detector is needed. The ring resonator with a SiGe- or Ge-doped region is coupled with a waveguide as shown in Fig. 48 [101]. The optical signal with the same wavelength as the ring's resonance frequency is coupled into the ring. The coupled optical signal is absorbed by the SiGe region and transformed to current.

Finally, a photonic switch is a major building component to route optical data from a source to a destination. (A photonic switch is not shown in Fig. 48.) A photonic switch can be built in several ways. One of the ways is using a set of ring resonators and waveguides. The important point is that characteristics of the ring resonator used in a photonic switch are different from those of the ring resonator used in a modulator and a detector.

To support WDM transmissions, a ring resonator used in a photonic switch should have multiple resonance frequencies. This broadband ring resonator has the resonance profile of Fig. 50. For example, in Fig. 51 the broadband ring resonator provides two resonant wavelengths, so the photonic switch can change the route of the two optical signals at a time.

To build our mesh hybrid network system, we actually need higher-order photonic switches than the simple one of Fig. 51. The number of in/out ports required in our mesh

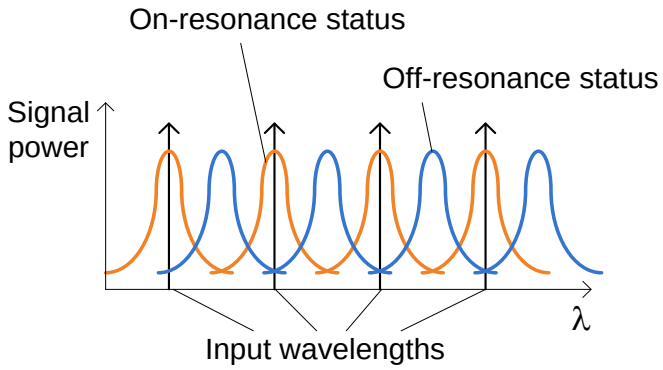


Figure 50. Resonance profile of a broadband ring resonator.

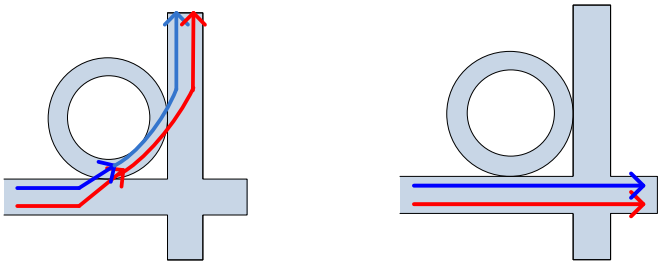


Figure 51. Simple photonic switch.

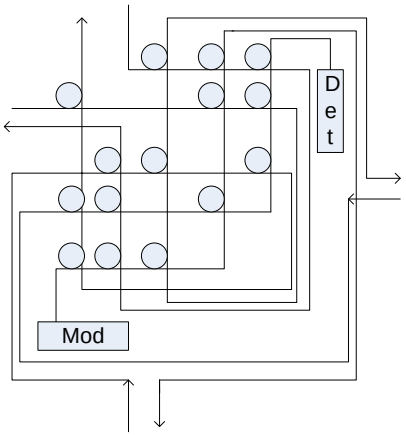


Figure 52. 5 x 5 non-blocking photonic switch [99].

hybrid system is five – four for E, W, S, N, and the other for a local PE. The Columbia lab developed a non-blocking 5 x 5 photonic switch by using a set of ring

resonators and waveguides as shown in Fig. 52. It provides a fully non-blocking feature at the cost of more number of ring resonators and higher insertion loss (than their first type switch that exhibits blocking – this switch is not shown here). We use this photonic switch at every node in the hybrid mesh network system.

4.3.6 Optical device parameters: insertion loss and energy

4.3.6.1 Insertion loss

As in electrical systems, optical systems have a dynamic power range in that the correct functionality is guaranteed at the system level - i.e., to achieve reliable optical communications , a signal power level should be within the range. The upper bound is the threshold that non-linearity of photonic devices starts being induced. The upper bound is determined by maximal input power allowed in the modulator at the source node. The lower bound is determined by the sensitivity of the photo detector at the final destination node.

An initial source of optical signals is the laser at the source node as shown in Fig. 48. Beginning at that point, optical signals experience power loss (called insertion loss in the optical world) along the propagation path due to several different factors. A main source of insertion loss is shown in Table 4. The first type of insertion loss is related to a waveguide; optical signals fundamentally experience propagation loss along a waveguide [measured in dB/cm]. In addition to a straight waveguide, bending of a waveguide and

Table 4. Insertion loss [99].

Factor	Insertion loss
Ring – off state	~=0
Ring – on state	0.5
Waveguide propagation	1.5dB/cm
Waveguide bending	0.005dB/90 ^{o}
Waveguide crossing	0.05

crossing of waveguides are needed to build optical network systems. The bending and crossing are another source of insertion loss related to a waveguide. The second type of insertion loss is related to a resonant ring that is a building component of modulators, detectors, and broadband switches. Insertion loss is different in the on-resonance and off-resonance status. In WDM-based systems, this insertion loss affects system performance in a very critical way that determines the number of wavelengths transmitted simultaneously through a single WDM waveguide. In theory, system bandwidth is linearly proportional to this number of wavelengths.

We explain how to calculate the number of simultaneously transmittable wavelengths in WDM-based systems by insertion loss analysis. The dynamic range described above is used to calculate it. The power level received at the photo detector of the final destination node must be greater than the sensitivity of the detector. Considering the worst-case (biggest) optical loss, each wavelength should be generated by laser emitters to have enough power - i.e., (each wavelength's power ─ the worst-case loss) ≥ the sensitivity of the detector. Note that the power of every individual wavelength should be considered here since a single detector can observe only one wavelength.

Another limitation is the upper bound of the optical power range. (Note that mutiple wavelengths are generated by a single laser emitter.) The sum of all wavelengths' power must be below the threshold that causes non-linear activities of photonic devices. In this way, the number of wavelengths can be calculated.

4.3.6.2 Energy consumption of optical devices

To measure power consumption of the hybrid optical system, we need energy consumption parameters of every photonic device. Those parameters are provided by Columbia lab [99]. They got those numbers by measuring real and contemporary photonic devices. We use these parameters to measure power consumption by embedding

these into PhoenixSim – an interconnection network simulator that will be explained in Section 4.3.8.

4.3.7 FFT application

Fast Fourier transform (FFT) is a very popular kernel application in DSP and communication fields – e.g., spectral analysis, interpolation, filtering, and multi-carrier modulation (such as OFDM). Fourier Transform (FT) is a mathematical method to transform a time-domain signal to a frequency-domain signal, which is generally applied to continuous-time signals. Discrete Fourier transform (DFT) is a discrete version that can work on discrete-time signals. This discrete version is essential since FT is mostly computed by (digital) computers where only digital data is accepted. But computing DFT according to its mathematical definition is too complex to be practical. FFT provides a fast way to compute DFT – the computational complexity of DFT is $O(N^2)$, but that of FFT is $O(N\log N)$.

There are several different FFT algorithms. Among them, the Cooley-Tukey algorithm proposed by J. W. Cooley and J. W. Tukey [102] is the most common one. It is a divide-and-conquer algorithm – DFT of a composite size $N = N_1N_2$ can be computed

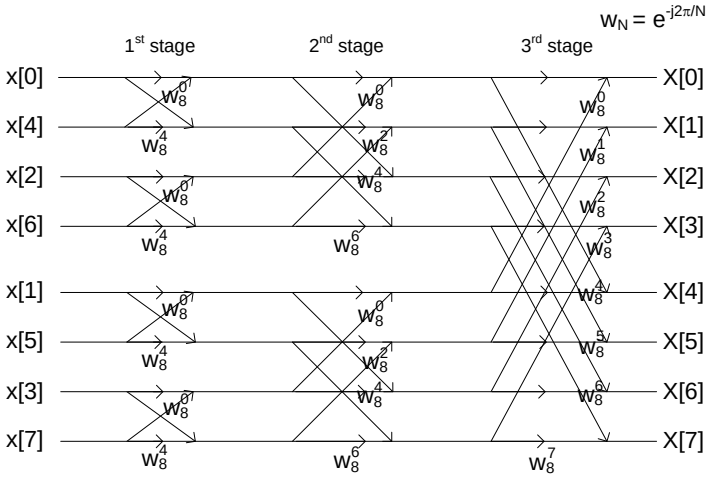


Figure 53. Radix-2 8-point Cooley-Turkey FFT.

using two DFTs of smaller size N_1 and N_2 where N_1 and N_2 are data sets that are interleaved each other – e.g., N_1 is the even members of the N data set and N_2 is the odd members. This step can be iteratively applied until the resulting data set is reduced to two data values.

Figure 53 shows an example of the Cooley-Turkey algorithm for radix-2 8-point FFT. The Cooley-Turkey algorithm can be applied to any composite length; as an example, the composite length of $N = 2^X$ (x is any positive integer number) is shown here, which is called a radix-2 Cooley-Turkey algorithm. In Fig. 53, eight input data (x[0] to x[7]) is divided into two subsets that are interleaved each other – x[0], x[2], x[4], x[6] in the first subset and x[1], x[3], x[5], x[7] in the other subset. Then each subset is divided into its own two subsets further – x[0], x[4] in one subset, x[2], x[6] in the second subset, x[1], x[5] in the third subset, and x[3], x[7] in the final subset. As a result, every subset is 2-point FFT.

For our purpose of experiments, we do not fully implement the FFT, instead we embed execution times spent on each stage into the PhoenixSim simulator [103]. The execution times are provided by [104] at various platforms.

4.3.8 Evaluations – both electrical and hybrid mesh network systems

In this section, we measure throughput and power consumption of the Cooley-Turkey FFT algorithm and three synthetic benchmarks on the three switching schemes. And we compute power efficiency [in samples/sec/W].

For the performance evaluations, we use PhoenixSim (Photonic and Electrical Network Integration and Execution Simulator) that has been developed in C++ by Columbia lab. PhoenixSim is an event-driven simulator built on OMNET++ environments [105]. PhoenixSim captures detailed physical characteristics of photonic and electrical components.

Table 5. Simulation parameters.

	Emesh-PS	Emesh-CS	Pmesh-CS
NoC parameters			
# of nodes	64 (8x8)		
clockRate_cntrl (GHz)	1.6	1.0	1.0
clockRate_data (GHz)	-	2.5	2.5
Electrical Channel Width (bits)	64	32 (ctrl path) & 128 (data path)	32
routerBufferSize per virtual channel (bits)	1024	128	128
# of virtual channels per port	2	1	1
DRAM parameters			
DRAM type	SDRAM		
DRAM freq. (MHz)	1066		
# of ranks per MAP	2		
Chips Per DIMM	4		
# of banks per chip	4		
# of rows per bank	8192		
# of columns per bank	512		
Transaction scheduling policy	First-come first-served		
Row buffer management policy	Close page		
Addr mapping policy	SDRAM base map		
Refresh policy	All ranks and all banks at a time		
FFT Application parameters			
# of points	2^{10} , 2^{12} , 2^{14}		
Data resolution	32 bits		

Simulation parameters are set as shown in Table 5. In addition to the FFT application, we measure power efficiency on three synthetic benchmarks – random, neighbor, and tornado [18] – with two different message sizes as shown in Table 6. The simulation results are shown in Fig. 54.

As shown in Fig. 54, the electrical mesh supporting packet switching shows the best power efficiency in the FFT application. This is due to the fact that the data size transferred per connection is generally very small in FFT – we fix the size as 32 bits in our simulations. Therefore, our circuit switching-based systems - electrical mesh and hybrid mesh systems based on circuit switching – are not appropriate for the Cooley-

Table 6. Synthetic benchmarks.

	Random		Neighbor		Tornado	
Message size (B)	64	12800	64	12800	64	12800
# of messages	6400	6400	6400	6400	6400	6400

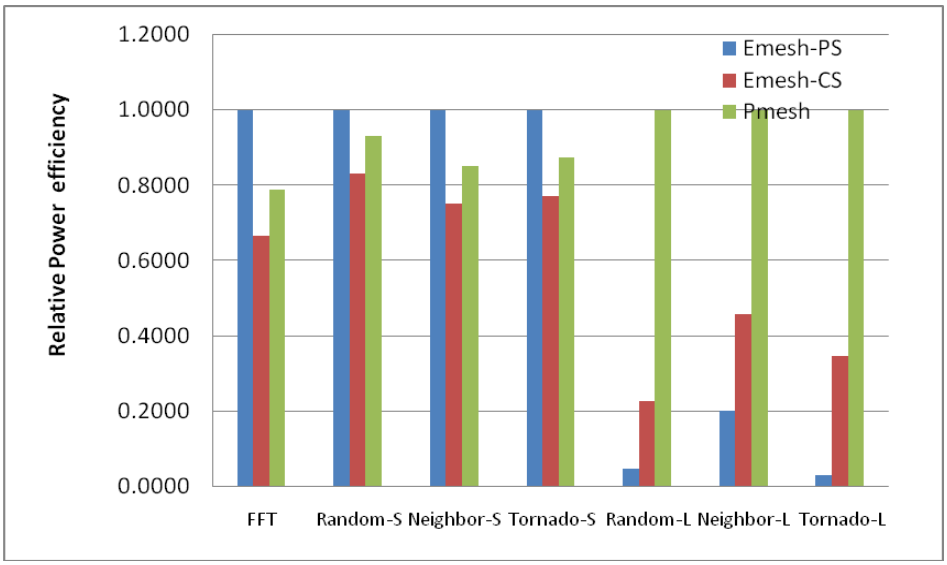


Figure 54. Power efficiency.

Turkey FFT algorithm. In other words, path setup and teardown overheads overshadow the positive effect of dedicated resources.

A similar pattern is observed for all synthetic benchmark with a small message size - . But in the case of a large message size, the hybrid optical network shows the best power efficiency at all three benchmarks. Based on these experiments, the message size is a very critical parameter to determine whether or not the hybrid photonic network is appropriate to be applied.

4.3.9 Summary

In Sections 4.2 and 4.3, we designed network-centric FFT processors. We consider the electrical mesh network based on packet switching and circuit switching respectively.

And we apply a different medium as interconnection network; we consider the hybrid mesh network based on circuit switching. The hybrid network is an electrically-controlled photonic network.

We map the Cooley-Turkey FFT algorithm onto the three NoC systems, then measure the power efficiency using PhoenixSim at various input sample sizes. The simulation results show that in the FFT algorithm, the packet-switching electrical network provides the best power efficiency regardless of the input sample size. This is due to the fact that the data transfer size per connection is very small in the FFT algorithm. In this environment, circuit switching does not provide its advantage (a dedicated end-to-end path). To show the effect of a packet size on power efficiency, we performed further experiments on the three synthetic benchmarks. According to the experiment results, a message size is a very critical parameter to determine whether or not the hybrid photonic network is appropriate to be used.

CHAPTER V

CONCLUSIONS

In this thesis, we propose new methods for designing high-performance embedded computer system architectures.

First, we proposed architectural enhancements to GPUs for the Turbo decoding algorithm, one of the popular communication applications. Experimental results by our cycle-accurate GPU simulator demonstrate that increasing the number of SMs and supporting the sub-word parallelism are the most promising ways to improve throughput. We achieved the 2 Mbps requirement of the WCDMA standard with all the architectural enhancements on the four-SM GPU (1.9 Mbps) or on the six-SM GPU (2.6 Mbps), while without any enhancements at least 12 SMs are required. An area overhead, however, by increasing the number of SMs is most severe. We also showed the estimated area overhead due to architectural enhancements. When an area limitation is given as a design parameter, we should select some of the architectural enhancements based on our throughput-area trade-off analysis.

Secondly, we proposed a high-level compiler technique to improve system throughput in DSP multi-processor systems by exploiting useful features of contemporary DRAM main memory systems. Our high-performance buffer mapping policy is for SDF-based DSP applications that are targeted to multiprocessor systems supporting the shared-memory programming model. The proposed policy exploits bank and rank concurrency of contemporary DRAM main memory systems according to the careful memory system modeling and parallelism analysis. We use a graph coloring technique to analyze the IPC (inter-processor communication) edge-level parallelism and a 1-bit predictor to analyze the comm/sync-level parallelism. In our experiments, we measured application throughput on both synthetic and real benchmarks. The simulation results show that the

proposed buffer mapping policy enhances throughput, especially in memory-intensive applications with relatively small total execution time of actors. Whether or not the even2 (comm/sync-level) mapping approach is used should be determined based on sync_read prediction accuracy, which can be estimated at compile-time. The performance improvements of the proposed buffer mapping policy are achieved in general at the cost of additional banks and bank under-utilization.

Finally, we proposed the parallel Turbo decoder and the parallel FFT processors. The parallel Turbo decoder is based on the VOQ input-queued crossbar switch, which can accommodate arbitrary interleaving/deinterleaving patterns during run-time. The proposed architecture dynamically resolves both internal and external conflicts of network resources. And we found that the PIM algorithm can be applied to Turbo decoding. By using our cycle-accurate simulator, we proved that our network-centric Turbo decoder can satisfy throughput requirements of all contemporary communication standards. We designed NoCs for another embedded kernel application – FFT. The electrical mesh network supporting packet switching or circuit switching, and the hybrid mesh network supporting circuit switching were considered. We mapped the Cooley-Turkey FFT algorithm onto those three network systems and measured power efficiency by using PhoenixSim. According to our experiments, the packet-switching electrical mesh network provides better power efficiency than the other two network systems. This is due to the fact that a data transfer size per connection is very small in the FFT algorithm. By further experiments on synthetic benchmarks, we found that a message size is a very critical parameter to determine whether or not the hybrid photonic network is appropriate to be applied.

REFERENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits," Electronics, vol. 38, no. 8, Apr. 1965.

[2] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy, "POWER4 system microarchitecture," IBM Journal of R&D, vol. 46, no. 1, pp. 5–26, 2002.

[3] W. J. Ebel, "Turbo-code implementation on C6x," Technical report, Alexandria Research Institute-Virginia Polytechnic Institute and State University, 1999.

[4] F. Kienle, H. Michel, F. Gilbert, and N. When, "Efficiet MAP-algorithm implementation on programmable architectures," Advances in Radio Science, vol. 1, pp. 259–263, 2003.

[5] Samsung Semiconductor, FPM DRAM 4M x 16 part no. KM416V4100C, http://www.usa.samsungsemi.com/products/prodspec/dramcomp/KM416V40(1)00 C.PDF, 1998.

[6] IBM, EDO DRAM 4M x 16 part no. IBM0165165PT3C, http://www.chips. ibm.com/products/ memory/88H2011/88H2011.pdf, 1998.

[7] IBM, SDRAM 1M x 16 x 4 bank part no. IBM0364164, http://www.chips. ibm.com/products/ memory/19L3265/19L3265.pdf, 1998.

[8] Rambus, 16/18Mbit & 64/72-Mbit concurrent RDRAM data sheet, http://www.ram bus.com/docs/Cnctds.pdf, 1998.

[9] Rambus, Direct RDRAM 64/72-Mbit data sheet, http://www.rambus.com/docs/ 64dDDS.pdf, 1998.

[10] Process integration, devices & structures, International Technology Roadmap for Semiconductors, 2007.

[11] S. Raoux et al., "Phase-change random access memory: A scalable technology," IBM Journal of Research and Development, vol. 52, no. 4.5, pp. 465−479, July 2008.

[12] S. Rixner, "Memory controller optimizations for web servers," In Proceedings of the International Symposium on Microarchitecture, pp. 355–366, 2004.

[13] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," In Proceedings of the International Symposium on Computer Architecture, pp. 128–138, 2000.

[14] R. Raghavan and J. Hayes, "On randomly interleaved memories," In Proceedings of Supercomputing, pp. 49–58, 1990.

[15] J. Corbal, R. Espasa, and M. Valero, "Command vector memory systems: High performance at low cost," In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, pp. 68–77, 1998.

[16] B. K. Mathew, S. A. McKee, J. B. Carter, and A. Davis, "A design of a parallel vector access unit for SDRAM memory systems," In Proceedings of the International Symposium on High-Performance Computer Architecture, pp. 39–48, 2000.

[17] V. E. Benes, Mathematical theory of connecting network and telephone traffic, New York, NY: Academic, 1965.

[18] W. J. Dally, "Performance analysis of k-ary n-cube interconnection networks," IEEE Transactions on Computers, vol. 39, no. 6, pp. 775–785, 1990.

[19] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb, "iWarp: An integrated solution to high-speed parallel computing," In Proceedings of Supercomputing, Nov. 1988.

[20] R. E. Kessler and J. L. Schwarzmeier, "Cray T3D: a new dimension for Cray research," In Proceedings of COMPCON, pp. 176–182, Feb. 1993.

[21] S. Scott and G. Thorson, "The Cray T3E network: Adaptive routing in a high performance 3D Torus," In Proceedings of Hot Interconnects IV, pp. 147−156, Aug. 1996.

[22] J. W. Goodman, F. J. Leonberger, S.-Y. Kung, and R. A. Athale, "Optical interconnections for VLSI systems," Proceedings of the IEEE, vol. 72, pp. 850–866, 1984.

[23] N. Kirman, M. Kirman, R. K. Dokania, J. F. Martinez, A. B. Apsel, M. A. Watkins, and D. H. Albonesi, "Leveraging optical technology in future bus-based chip multiprocessors," In Proceedings of the International Symposium on Microarchitecture, pp. 492–503, Dec. 2006.

[24] Y. Pan, P. Kumar, J. Kim, G. Memik, Y. Zhang, and A. Choudhary, "Firefly: Illuminating future network-on-chip with nanophotonics," In Proceedings of the International Symposium on Computer Architecture, pp. 429–440, June 2009.

[25] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. P. Jouppi, M. Fiorentino, A. Davis, N. Binkert, R. G. Beausoleil, and J. H. Ahn, "Corona: System implications of emerging nanophotonic technology," In Proceedings of the International Symposium on Computer Architecture, pp. 153–164, 2008.

[26] D. Brunina, C. P. Lai, A. S. Garg, and K. Bergman, "Optically-connected memory systems for high-performance computing," 21st Annual Workshop on Interconnections within High Speed Digital Systems, May 2010.

[27] D. Brunina, A. S. Garg, H. Wang, C. P. Lai, and K. Bergman, "Experimental demonstration of optically-connected SDRAM," Photonics in Switching 2010 PMC5, July 2010.

[28] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," IEEE Micro, vol. 28, no. 2, pp. 39–55, 2008.

[29] CUDA Programming guide v.2.3.1, http://www.nvidia.com/object/cuda_develop.html.

[30] http://en.wikipedia.org/wiki/Turbo_code.

[31] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," IEEE Transactions on Information Theory, vol. IT-20, pp. 284–287, Mar. 1974.

[32] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," In Proceedings of the International Conference on Communications, 1995.

[33] J. A. Erfanian, S. Pasupathy, and G. Gulak, "Reduced complexity symbol detectors with parallel structures for ISI channels," IEEE Transactions on Communications, vol. 42, pp. 1661–1671, 1994.

[34] S. J. Lee, N. R. Shanbhag, and A. C. Singer, "Area-efficient high throughput MAP decoder architectures," IEEE Transactions on Very Large Scale Integration Systems, vol. 13, no. 8, pp. 921–933, Aug. 2005.

[35] 3rd Generation Partnership Project (3GPP) technical specification group: radio access network, multiplexing and channel coding (FDD), Release 1999, http://www.3gpp.org.

[36] M. Marandian, J. Fridman, Z. Zvonar, and M. Salehi, "Performance analysis of turbo decoder for 3GPP standard using the sliding window algorithm," In Proceedings of the International Symposium on Personal, Indoor and Mobile Radio Communications, vol. 2, pp. e127–e131, Sep. 2001.

[37] Ocelot project site, http://code.google.com/p/gpuocelot.

[38] E. Boutillon, C. Douillard, and G. Montorsi, "Iterative decoding of concatenated convolutional codes: Implementation issues," In Proceedings of the IEEE Custom Integrated Circuits Conference, pp. 1201–1227, 2007.

[39] R. B. Lee, “Subword parallelism with MAX-2,” IEEE Micro, pp. 51–59, July/Aug. 1996.

[40] D. Lee, M. Wolf, and H. Kim, “Design space exploration of the Turbo decoding algorithm on GPUs,” In Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Oct. 2010.

[41] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, and C. Chakrabarti, “SODA: A low-power architecture for software radio,” In Proceedings of the International Symposium on Computer Architecture, pp. 89–101, 2006.

[42] W. C. Hasenplaugh and M. A. Neifeld, “Image binarization techniques for correlation-based pattern recognition,” Optical engineering, vol. 38, p. 1907, 1999.

[43] N. Weste and D. Harris, CMOS VLSI design: A circuits and systems perspective, Addison Wesley, 2004.

[44] P. Asadi and K. Navi, “A novel high-speed 54 x 54 bit multiplier,” American Journal of Applied Sciences, vol. 4, pp. 666-672, 2007.

[45] http://en.wikipedia.org/wiki/NVIDIA_GPU.

[46] Y. Lin, S. Mahlke, T. Mudge, C. Chakrabarti, A. Reid, and K. Flautner, “Design and implementation of turbo decoders for software defined radio,” In Proceedings of IEEE Workshop on Signal Processing System Design and Implementation, pp. 22–27, 2006.

[47] Micron, Micron SDRAM 256MB data sheet, http://download.micron.com/pdf/ datasheets/ dram/ sdram/256MSDRAM.pdf.

[48] Elpida, Elpida SDRAM 256MB data sheet, http://www.elpida.com/pdfs/ E0984E20.pdf.

[49] S. Sriram and S. S. Bhattacharyya, Embedded multiprocessors: Scheduling and synchronization, CRC Press, 2009.

[50] E. A. Lee and D. G. Messerschmitt, “Synchronous dataflow,” Proceedings of the IEEE, pp. 1235–1245, 1987.

[51] M. Ade, R. Lauwereins, and J. A. Peperstraete, “Buffer memory requirements in DSP applications,” In Proceedings of the International Workshop on Rapid System Prototyping, pp. 108–123, 1994.

[52] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, “Converting graphical DSP programs into memory-constrained software prototypes,” In Proceedings of the International Workshop on Rapid System Prototyping, pp. 194–200, 1995.

[53] M. R. Garey and D. S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, W. H. Freeman and Company, New York, 1999.

[54] S. S. Bhattacharyya, S. Sriram, and E. A. Lee, "Optimizing synchronization in multiprocessor DSP systems," IEEE Transactions on Signal Processing, vol. 62, pp. 1605–1618, June 1997.

[55] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, "DRAMsim: A memory-system simulator," SIGARCH Computer Architecture News, vol. 33, no. 4, pp. 100–107, Nov. 2005.

[56] E. Cooper-Balis and B. Jacob, "Fine-grained activation for power reduction in DRAM," IEEE Micro, vol. 30, no. 3, pp. 34–47, May 2010.

[57] N. Rafique, W. T. Lim, and M. Thottethodi, "Effective management of DRAM bandwidth in multicore processors," In Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pp. 245–258, 2007.

[58] B. Khargharia, S. Hariri, and M. S. Yousif, "An adaptive interleaving technique for memory performance-per-watt management," IEEE Transactions on Parallel and Distributed Systems, vol. 20, no. 7, pp. 1011–1022, July 2009.

[59] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling," IEEE Computer, vol. 35, no. 2, pp. 59–67, Feb. 2002.

[60] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," SIGARCH Computer Architecture News, vol. 33, no. 4, pp. 92–99, Nov. 2005.

[61] B. Jacob, DRAMsim: A detailed memory-system simulation framework, http:// www.ece.umd.edu/ dramsim/v1/#download.

[62] R. P. Dick, D. L. Rhodes, and W. Wolf, "Tgff: task graphs for free hardware/software codesign," In Proceedings of the International Workshop on CODES/CASHE, pp. 97–101, 1998.

[63] C. L. McCreary, A. A. Kahn, J. Thompson, and M. E. McArdle, "A comparison of heuristics for scheduling DAGS on multiprocessors," In Proceedings of International Parallel Processing Symposium, pp. 446–451, 1994.

[64] D. Esteban and C. Galand, "Application of quadrature mirror filter to split-band voice coding schemes," In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, pp. 191–195, 1977.

[65] K. Karplus and A. Strong, "Digital synthesis of plucked string and drum timbres," Computer Music Journal, vol. 7, no. 2, pp. 43–55, 1983.

[66] P. K. Murthy and S. S. Bhattacharyya, "Buffer merging – a powerful technique for reducing memory requirements of synchronous dataflow specifications," ACM Transactions on Design Automation of Electronic Systems, vol. 9, no. 2, pp. 212–237, 2004.

[67] M. Ade, R. Lauwereins, and J. A. Peperstraete, "Data memory minimization for synchronous dataflow graphs emulated on DSP-FPGA targets," In Proceedings of the Design Automation Conference, pp.64–69, 1997.

[68] J. Zhu, I. Sander, and A. Jantsch, "Buffer minimization of real-time streaming applications scheduling on hybrid CPU/FPGA architectures," In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, pp. 1506–1511, 2009.

[69] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi, "Throughput analysis of synchronous dataflow graphs," In Proceedings of the International Conference on Application of Concurrency to System Design, pp. 25–36, 2006.

[70] H. Kee, S. S. Bhattacharyya, and J. Kornerup, "Efficient static buffering to guarantee throughput – optimal FPGA implementation of synchronous dataflow graphs," In Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, pp. 136–143, 2010.

[71] S. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," In Proceedings of the Design Automation Conference, pp. 899–904, 2006.

[72] E. Boutillon, C. Douillard, and G. Montorsi, "Iterative Decoding of Concatenated Convolutional Codes: Implementation Issues," Proceedings of the IEEE, vol. 95, no. 6, pp. 1201–1227, 2007.

[73] Y. Zheng and Y. T. Su, "A new interleaver design and its application to turbo codes," in Proc. IEEE Vehic. Tech. Conf. (VTC'02), pp. 1437–1441, 2002.

[74] C. Berrou, Y. Saouter, C. Douillard, S. Keroudan, and M. Jezequel, "Designing good permutations for turbo codes: towards a single model," in Proc. IEEE Int. Conf. on Commun. (ICC'04), pp. 341–345, 2004.

[75] A. Nimbalker, T. K. Blankenship, B. Classon, T. E. Fuja, and D. J. Costello, "Contention-free interleavers for high-throughput Turbo decoding," IEEE Trans. on Communications, vol. 56, no. 8, pp. 1258-1267, 2008.

[76] C. Neeb, M. J. Thul, and N. Wehn, "Network-on-chip-centric approach to interleaving in high throughput channel decoders," in Proc. IEEE Int. Symp. on Circuits and Systems (ISCAS'05), vol. 2, pp. 1766 – 1769, 2005.

[77] H. Moussa, O. Muller, A. Baghdadi, and M. Jézéquel, "Butterfly and Benes-based on-chip communication networks for multiprocessor Turbo decoding," in Proc. Design, Automation and Test in Europe (DATE'07), pp. 654-659, 2007.

[78] M. Karol, M. Hluchyj, and S. Morgan, "Input versus output queueing on a space-division switch," IEEE Trans. Communications, vol. 35, no. 12, pp.1347-1356, 1987.

[79] Y. Tamir and G. Frazier, "High-performance multi-queue buffers for VLSI communication switches," in Proc. Int. Symp. on Comp. Arch. (ISCA'88), pp.343-354, 1988.

[80] T. Anderson, S. Owicki, J. Saxe, and C. Thacker, "High speed switch scheduling for local area networks," ACM Trans. on Computer Systems, pp. 319-352, 1993.

[81] J. E. Hopcroft and R. M. Karp, "An n5/2 algorithm for maximum matching in bipartite graphs," Society for Industrial and Applied Mathematics J. Comput., pp. 225-231, 1973.

[82] R. E. Tarjan, "Data structures and network algorithms," Society for Industrial and Applied Mathematics, Pennsylvania, 1983.

[83] N. McKeown; "Scheduling algorithms for input-queued cell switches," PhD Thesis, University of California, Berkeley, 1995.

[84] D. Lee and W. Wolf, "Power- and area-efficient single SISO architecture of Turbo decoder," Technical report, GIT-CERCS-09-10, 2009.

[85] G. Nong, J. K. Muppala, and M. Hamdi, "Analysis of nonblocking ATM switches with multiple input queues," IEEE/ACM Transactions on Networking, vol. 7, no. 1, pp. 60-74, 1999.

[86] Y. Sun, Y. Zhu, M. Goel, and J. R. Cavallaro, "Configurable and scalable high throughput turbo decoder architecture for multiple 4G wireless standards," in Proc. Int. Conf. on Application-Specific Systems, Architectures and Processors (ASAP'08), pp. 209-214, 2008.

[87] B. Bougard et al., "A scalable 8.7-nJ/bit 75.6-Mb/s parallel concatenated convolutional (turbo-) codec," in Proc. IEEE Int. Solid-State Circuit Conf. (ISSCC'03), 2003.

[88] Y. Lin et al., "Design and implementation of turbo decoders for software defined radio," in Proc. IEEE Workshop on Signal Processing System Design and Implementation (SiPS'06), pp. 22–27, 2006.

[89] P. Ituero and M. Lopez-Vallejo, "New schemes in clustered vliw processors applied to turbo decoding," in Proc. Int. Conf. on Application-Specific Systems, Architectures and Processors (ASAP'06), pp. 291–296, 2006.

[90] W. J. Dally and B. Towles, Principles and practices of interconnection networks, Morgan Kaufmann Publishers, 2004.

[91] W. J. Dally, "Virtual channel flow control," IEEE Transactions on Parallel and Distributed Systems, pp. 194-205, 1992.

[92] J. D. Day and H. Zimmerman, "The OSI reference model," In Proceedings of the IEEE, vol. 71, pp. 1334–1340, 1983.

[93] A. Lattes, H. A. Haus, F. J. Leonburger, and E. Ippen, "An ultrafast all optical gate," IEEE Journal of Quantum Electronics, vol. 19, no. 11, pp. 1718-1723, 1983.

[94] P. R. Prucnal, D. J. Blumenthal, and P. A. Perrier, "Self-routing photonic switching demonstration with optical control," Optical Engineering, vol. 26, no. 5, pp. 473-477, 1987.

[95] A. Shacham, K. Bergman, and L. P. Carloni, "Photonic networks-on-chip for future generations of chip multiprocessors," IEEE Transactions on Computers, vol. 57, no. 9, pp. 1246–1260, 2008.

[96] M. Petracca, B. G. Lee, K. Bergman, and L. Carloni, "Design exploration of optical interconnection networks for chip multiprocessors," In Proceedings of the IEEE Symposium on High Performance Interconnects, pp. 31-40, 2008.

[97] Y. Pan, P. Kumar, J. Kim, G. Memik, Y. Zhang, and A. Choudhary, "Firefly: illuminating future network-on-chip with nanophotonics," In Proceedings of the ISCA, pp. 429-440, 2009.

[98] H. Ishio, J. Minowa, K. Nosu, "Review and status of wavelength-division-multiplexing technology and its application," IEEE Journal of Lightwave Technology, vol. 2, no. 4, pp. 448-463, 1984.

[99] G. Hendry, E. Robinson, V. Gleyzer, J. Chan, L. P. Carloni, N. Bliss, K. Bergman, "Circuit-switched memory access in photonic interconnection networks for high-performance embedded computing," In Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-12, 2010.

[100] A. Gubenko, I. Krestnikov, D. Livshtis, S. Mikhrin, A. Kovsh, L. West, C. Bornholdt, N. Grote, and A. Zhukov, "Error-free 10 gbit/s transmission using individual fabry-perot modes of low-noise quantum-dot laser," Electronic Letters, vol. 43, no. 25, pp. 1430–1431, 2007.

[101] O. I. Dosunmu, D. D. Cannon, M. K. Emsley, L. C. Kimerling, and M.S. Unlu, "High-speed resonant cavity enhanced Ge photodetectors on reflecting Si substrates for 1550-nm operation," IEEE Photonics Technology Letters, vol. 17, pp. 175-177, 2005.

[102] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex fourier series,” Mathematics of computation, vol. 19, no. 90, pp. 297–301, 1965.

[103] J. Chan, G. Hendry, A. Biberman, K. Bergman, and L. P. Carloni, “Phoenixsim: A simulator for physical-layer analysis of chip-scale photonic interconnection networks,” In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, pp. 691-696, 2010.

[104] FFTW Home Page, http://www.fftw.org.

[105] OMNeT++ Network Simulation Framework, http://www.omnetpp.org.