ABSTRACT

VASANTH ASOKAN, Relaxing Control Flow Constraints in ASTI (Under the direction of Dr. Alexander G. Dean)

Asynchronous Software Thread Integration (ASTI) provides methods to reclaim sub-bit duration idle intervals in the real-time threads which bang bits onto and off from the network bus, by allowing asynchronous progress of secondary threads. Current ASTI techniques constrain the secondary thread from having procedures and procedure calls because of conflicts in timing that arise. Constraining procedure calls is very inelegant and limits the usefulness of the techniques developed in ASTI. It also significantly lowers the ease with which software can be developed and maintained. Flat code is significantly larger in size compared to code that has been structured into procedures. This can be extremely important in embedded systems where code memory is at a premium.

This thesis presents a novel application of the traditional compiler optimization techniques - procedure inlining and procedure cloning, to remove the constraints on procedure calls with ASTI. Under the assumption that the system cannot tolerate the inlining of all procedure calls, this thesis provides an algorithmic approach to choosing some subset of calls to inline and to remove the timing conflicts in the remaining procedures in the most optimal fashion.

Timing conflicts in the remaining calls can be removed by padding calls to a procedure to the worst case. Procedure cloning can be used to extend this by clustering calls that have similar environments together and dedicating a clone of the procedure to the cluster. We show that this would result in significant reductions not only in the amount of static padding cycles required, but also in the amount executed at run-time too. We expect a secondary thread, which uses cloning to reduce the amount of padding, to show a significant speedup when compared to a non-cloned version. Providing a scientific approach for constraint relaxation, instead of an ad-hoc one, will also produce much more efficient code both in terms of size and consumed processor cycles.

RELAXING CONTROL FLOW CONSTRAINTS IN ASTI

By

Vasanth Asokan

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science

Computer Science

North Carolina State University

2003

Approved by

_____
Dr. Alexander G. Dean, Chairperson of the Advisory Committee

_____
Dr. Frank Mueller, Co-chair of Advisory Committee

_____
Dr. Mihail Sichitiu

Date
_____

BIOGRAPHY

Vasanth Asokan was born in Chennai, India. He lived most of his life in the Cotton City of India, Coimbatore. He graduated with a Bachelor's degree in Computer Science and Engineering from the College of Engineering, Guindy, Chennai in 2001.

He then came to North Carolina State University, seeking a Master's degree in Computer Science. Pursuing his interest in embedded and real-time systems, he joined the Center for Embedded Systems Research, where he had the opportunity to do exciting research in the refreshingly new area of STI.

ACKNOWLEDGEMENTS

I sincerely thank my advisor Dr. Alexander G. Dean, who has inspired me with his research to think differently and unconventionally. I thank my parents and my sister – they are all the motivation that I ever need in life.

TABLE OF CONTENTS

LIST OF FIGURES ..... **VI**
LIST OF TABLES ..... **VII**

CHAPTER 1
INTRODUCTION **1**

1.1 INTRODUCTION 1
1.2 MOTIVATION 2

CHAPTER 2
BACKGROUND **6**

2.1 SOFTWARE THREAD INTEGRATION 6
2.2 BIT-BANGED COMMUNICATION PROTOCOLS (BBCP) 7
2.3 ASTI 8

CHAPTER 3
RELATED RESEARCH **12**

3.1 PROCEDURE INLINING 12
3.2 PROCEDURE CLONING 14
3.3 CALL GRAPHS 15

CHAPTER 4
CONTRIBUTIONS: NEW METHODS **18**

4.1 APPLYING PROCEDURE INLINING AND CLONING 18
4.2 APPROACH FOR RELAXING CONSTRAINTS 20
4.3 PHASE ONE 21
4.3.1 Description *21*
4.3.2 Proposed technique *22*
4.3.3. Phase One Algorithm *24*
4.3.4 Observations about the algorithm *25*
4.4 PHASE TWO 25
4.4.1 Outline of Phase Two *25*
4.4.2 Partitioning Call Sites Efficiently *28*
4.4.3 Intervening Guest Code *32*
4.4.4 Call Graph Processing Order *33*
4.4.4 Multiple Primary Threads *35*
4.5 IMPLEMENTATION IN THRINT 36
4.5.1 Overview *36*
4.5.2 Code Transformations *37*

CHAPTER 5
EXPERIMENTS AND RESULTS **39**

5.1 BENCHMARK SUITE OF SECONDARY THREADS 39
5.2 METHODOLOGY OF EXPERIMENTS 41
5.3 RESULTS FOR SMART NODE APPLICATION 42
5.4 RESULTS FOR THE PID CONTROLLER APPLICATION 46
5.5 RESULTS FOR THE HOST INTERFACE THREAD 50
5.6 CODE EXPANSION DUE TO CLONING 53
5.7 SPEEDUP OVER NON-CLONED VERSIONS DUE TO CLONING 54

CHAPTER 6
CONCLUSIONS AND FUTURE WORK **57**

6.1 CONCLUSIONS 57
6.2 FUTURE WORK 57
REFERENCES 59
APPENDIX 1 61

LIST OF FIGURES

Page

Figure 1.1 Context switching through cocalls 2
Figure 1.2 The timing variability problem 3
Figure 2.1 Software thread integration 6
Figure 2.2 Idle time in primary thread 9
Figure 2.3 Building larger idle chunks in the primary thread 9
Figure 2.4 Integration 10
Figure 3.1 Sample call graph 16
Figure 4.1 Resolving conflicts 26
Figure 4.2 Plot of cycles left to insert next cocall for a sample procedure 27
Figure 4.3 Distance between two phases Δc_1 and Δc_2 30
Figure 4.4 Illustration of minimal spanning tree clustering algorithm 31
Figure 4.5 Sample call site ordering 33
Figure 5.1 Three types of hardware protocol controllers, and their replacement with a generic MCU 39
Figure 5.2 Smart node - Tradeoff between static padding cycles and clone ratio 44
Figure 5.3 Smart node - Cocall duration Vs static padding cycles 45
Figure 5.4 Representative procedure from Smart node – cocall duration Vs static padding cycles 46
Figure 5.5 PID controller - Tradeoff between static padding cycles and clone ratio 49
Figure 5.6 PID controller - Cocall duration Vs static padding cycles 49
Figure 5.7 Representative procedure from PID controller – cocall duration Vs static padding cycles 50
Figure 5.8 Host interface thread - Tradeoff between static padding cycles and clone ratio 52
Figure 5.9 Host interface thread - Cocall duration Vs static padding cycles 53
Figure 5.10 Increase in code expansion with changes in clone ratio 54
Figure 5.11 Speedup due to cloning 56

LIST OF TABLES

Table 4.1	Microcontroller configurations and high volume pricing	19
Table 4.2	Breakdown of automatic and manual tasks	36
Table 5.1	Smart node - Source code information	42
Table 5.2	Smart node - Phase one results	42
Table 5.3	Smart node - Phase one lining decisions	43
Table 5.4	PID controller - Source code information	46
Table 5.6	PID controller - Phase one results	47
Table 5.7	PID controller - Phase one inlining decisions	48
Table 5.8	Host interface thread - Source code information	50
Table 5.9	Host interface thread - Phase one results	51
Table 5.10	Host interface thread - Phase one inlining decisions	51

Chapter 1

Introduction

1.1 Introduction

Embedded communication networks are extensively used in a wide variety of applications. Software implementations of the communication protocols of embedded networks can replace dedicated network processors. Such hardware to software migration is very attractive for reasons relating to system cost, size, weight, power, time to develop and upgradeability. These software implementations are typically known as bit-banged communication protocols as the software directly drives signals onto and from the network bus.

Bit-banging implementations of communication protocols typically feature one or more threads which implement the access to the physical layer. These threads are characterized by fine grain idle time imposed upon them by the bit rate of the protocol. In most cases this idle time is too fine to be retrieved by conventional techniques for concurrency like context switching or interrupts.

Software Thread Integration (STI) [Dea00] targets low end embedded systems which cannot afford a high performance microprocessor yet need fine grain thread concurrency. Fine grain idle time is reclaimed using STI. Current STI techniques for integrating bit-banging implementations provide asynchronous progress of a *real-time guest (primary)* thread and a *host (secondary)* thread performing other useful work. These techniques are collectively referred to as Asynchronous Software Thread Integration (ASTI) [KSD03]. The basic idea behind ASTI is to *statically* schedule the real-time guest and host threads as co-routines. Co-routines enable the threads to execute cooperatively. The real-time thread yields execution to the host (secondary) thread at the start of suitable idle intervals. The secondary thread is structured so that there is a co-routine

call (cocall) back to the primary thread at the instants when the real-time work should start again. This is shown in *Figure 1.0.*

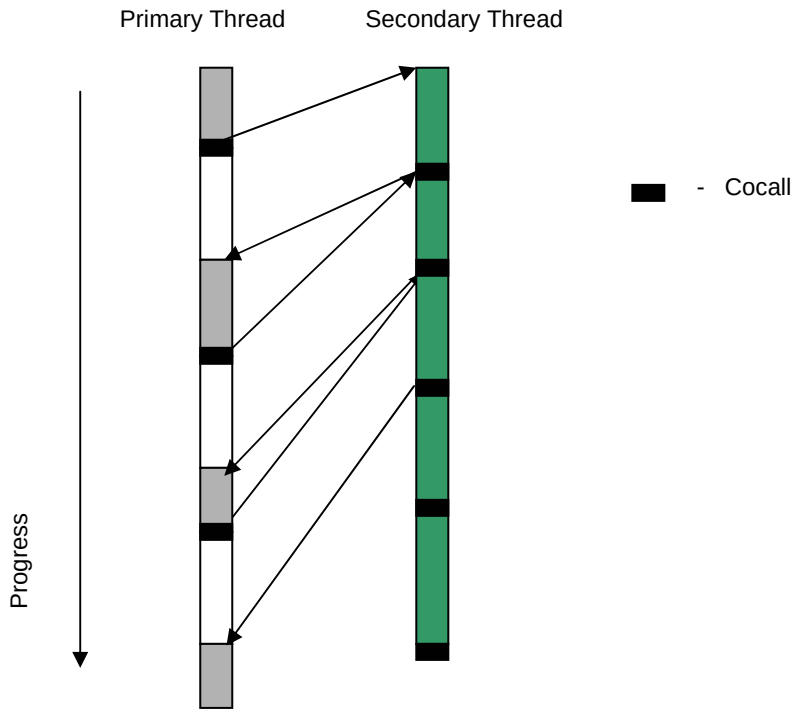


Fig. 1.1 *Context switching through cocalls*

1.2 Motivation

Current ASTI techniques apply a sequence of transformations on both the primary and the secondary threads to structure them as co-routines. The primary thread is initially transformed in such a fashion so as to expose *regular* and *constant* idle time intervals. This includes accumulating smaller idle intervals by removing *intervening guest code*, which gets integrated into the secondary thread at appropriate places. Co-routine calls are inserted at the start of the idle intervals in the primary thread. Correspondingly, co-calls are inserted throughout the control flow of the secondary thread at the same *regular*, *constant* intervals of $T_{SegmentIdle}$ (explained later). This will allow the secondary thread to execute in the idle intervals while transferring control back to the primary at the appropriate time.

Procedure calls in the secondary thread code result in the inapplicability of this approach of inserting cocalls at regular intervals. This is illustrated in *Figure 1.0*.

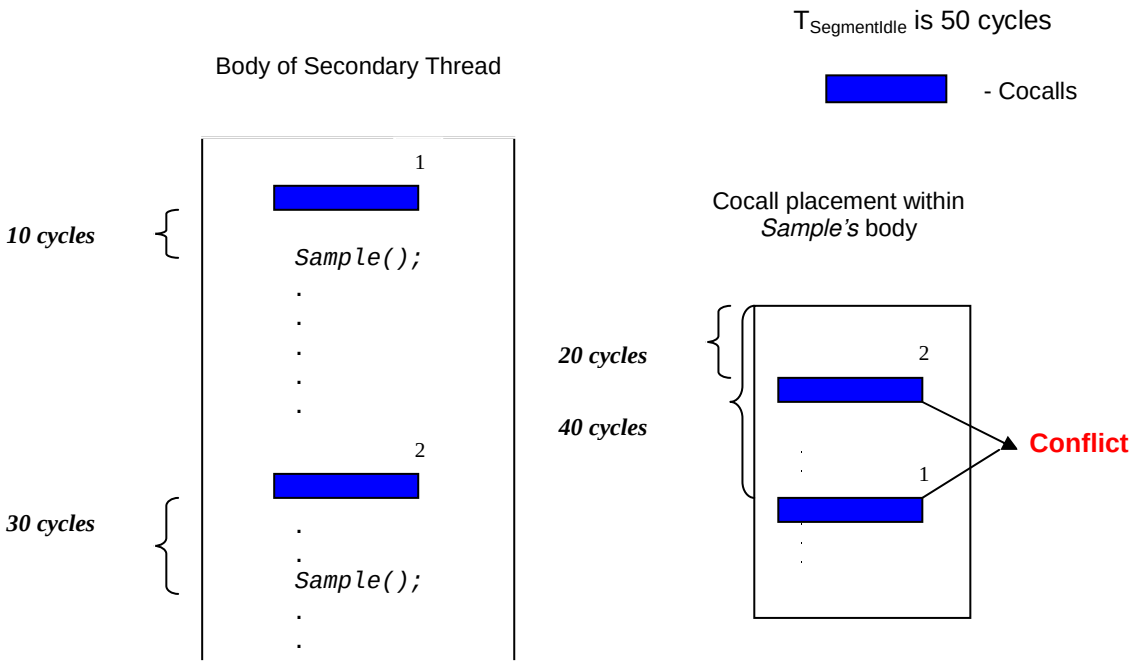


Fig. 1.2 *The timing variability problem.*

A call to a procedure will occur at a certain interval from the last cocall before it. This means that if there are multiple calls to a procedure then there is timing variability in the procedure's start time with respect to the last occurring cocall. Therefore, we cannot have a single copy of the procedure with cocalls inserted so as to possibly satisfy all the timing constraints that arise due to location of the various incoming calls. Referring to *Figure 2.0*, there are two calls to procedure *Sample* originating from two different points (1 and 2). Given a $T_{SegmentIdle}$ of 50 cycles, then corresponding to the first call, a cocall needs to be placed 40 cycles into the procedure. Corresponding to the second call site, a cocall needs to be placed 20 cycles into the procedure. Thus there are conflicting points at which the cocall must occur while executing the body of *Sample*. The same conflict arises while trying to place *intervening guest code* (explained later). The problem can be

further exacerbated by the presence of multiple primary threads, each having its own $T_{SegmentIdle}$. Therefore, ASTI currently constrains the secondary thread from having procedures and procedure calls. The secondary thread needs to be a flat body of code. Hence a developer wishing to employ STI needs to either write flat code or convert his structured code into a flat version before the STI phase.

If the code is procedurized, a straightforward solution would be to add padding cycles at each call site in such a fashion so as to pad the start of the procedure to the worst possible interval from last cocall. In the example in *Figure 1.0*, adding 20 cycles of padding before first call site would bring the relative offset from the last cocall of both the call sites to the same value (30). Therefore a single copy of the procedure body can now be used to insert cocalls to satisfy calls from both the call sites. The problem with this approach is that padding cycles represent no useful secondary thread work. Also the amount of padding added can get progressively worse when there are a lot of calls. The padding cycles added do not give an indication of the amount of cycles that are wasted dynamically, during run-time. For example, a call site which has been padded could be within a loop which runs for many iterations. Adding padding cycles is counterproductive to the main aim of STI which is to increase processor efficiency.

Constraining procedure calls is very inelegant and limits the usefulness of the techniques developed in ASTI. It also significantly lowers the ease with which software can be developed. Code written without modularization is extremely difficult to maintain and debug. This research is aimed at relaxing the control flow constraints imposed by the ASTI technique using traditional compiler optimizations like procedure cloning and procedure inlining.

Procedure inlining can be used to remove call sites by inlining a copy of the procedure body at the call site. Inlined procedures do not have the timing conflict problem with respect to cocalls as the code at the call site has been flattened after inlining. But inlining leads to code expansion which is a critical factor while designing embedded systems. This research aims at developing an approach that would remove procedure calls

while adhering to a code expansion limit, in the best possible fashion. For those procedures where inlining is not a viable alternative, call sites need to be padded to the worst case offset from last cocall and cocalls inserted into the procedure body. Procedure cloning can be applied to partition this set of call sites in such a fashion so as to minimize the amount of padding cycles added. We expect that there would be major reductions not only in the amount of static padding cycles required, but in their effects at run-time too. We expect a secondary thread which uses cloning to reduce the amount of padding, to show a significant speedup when compared to a non-cloned version.

Finally, allowing code writers to use structured code helps speed up the development time, leaving the constraint relaxation part to a later phase. Structured code is much easier to maintain, debug and re-use. The developer is also abstracted away from having to know details about the later STI phase and does not need to do anything special while developing and debugging his code. Providing a scientific approach to constraint relaxation, instead of an ad-hoc one, will produce much more efficient code both in terms of size and consumed processor cycles.

The ideas presented in this thesis have been applied on three different benchmark secondary threads – a host microcontroller to protocol microcontroller interface thread, a secondary thread implementing a smart node and a secondary thread implementing a closed loop PID control system. The tradeoffs between code size and wasted processor cycles, for each of these secondary threads, have been analyzed and an estimation of speedups due to cloning as against not cloning performed.

The thesis has been organized as follows. Chapter 2 details the background research that has been done with respect to STI and ASTI. Chapter 3 gives an overview of relevant research that has been conducted on procedure inlining and procedure cloning. Chapter 4 presents the new methods that have been developed to relax control flow constraints in ASTI. Chapter 5 discusses the experiments done and presents the analysis of the results obtained. Chapter 6 gives the conclusions and future work possible.

Chapter 2

Background

2.1 Software Thread Integration

STI [Dea00] is a compiler technology targeting embedded systems, which interleaves multiple assembly language threads to produce a single integrated version. This integrated thread offers low-cost concurrency, but still executes on a generic processor without fast context switches. STI can be used for hardware to software migration (HSM). HSM is the process of moving functions from dedicated hardware components to real-time software. HSM helps to improve system cost, size, weight, power, function availability, time to market, and field upgrades. STI targets low-end embedded systems which cannot afford the luxury of a high performance microprocessor yet require fine-grain thread concurrency. STI provides benefits in two scenarios. Firstly it can avoid the traditional context switch penalty, whenever it is too excessive, and hence allow threads to run relatively faster. Secondly, it can help reclaim idle time, which is generally filled up by executing busy wait loops, by executing instructions of another thread, which represents useful work. This is illustrated in *Figure 2.1*. A real-time guest thread, which features fine grain idle time, is chosen to be integrated with another existing thread, called the *host* thread.

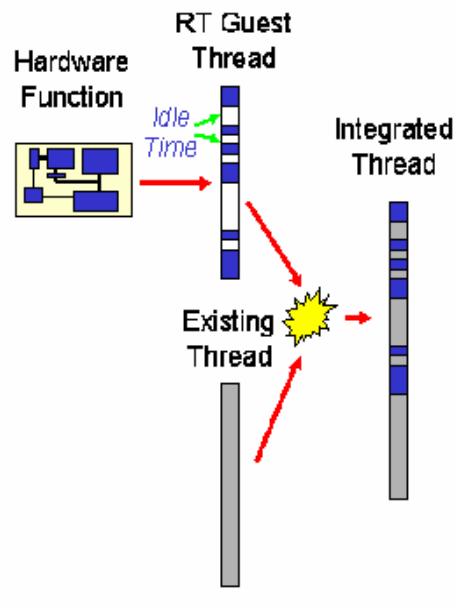


Fig 2.1 *Software thread integration*

STI uses a hierarchical program representation known as the *Control Dependence Graph* (CDG) [FOW87] which simplifies control flow analysis and the transformations

needed for integration. A CDG contains nodes for *code*, *predicates*, *loops* and *calls*. Vertical nesting indicates control dependence, while executing the CDG requires a left to right pre-order traversal. This hierarchical nature simplifies code motion and encapsulates timing information which is a critical requirement for the accurate placement of guest code at appropriate real time intervals. The graph is annotated with timing information by a static timing analysis phase (STA) which is performed bottom up through the levels of the CDG. STA also gives each node a best and worst case starting time based on the durations of the nodes on which this node is control dependant on. Code motion is then performed based on a set of transformations for moving guest sections into host code, predicates and loops, as well as integration of loops of different speeds etc. These code transformations are accompanied by a data-flow analysis phase which allows code to be moved freely. Register reallocation and partitioning techniques are also used to provide each thread being integrated with its own register set or minimize the conflicts between the sets.

Thrint is a post-pass (back-end) compiler that has been developed to automate various phases of STI [Dea03]. It reads in assembly code, performs control-flow, data-flow and static timing analysis, and integrates threads.

2.2 Bit-banged Communication Protocols (BBCP)

Software implementations of communication protocols are typically known as bit-banged protocols. They replace a dedicated hardware communication unit in software. Some examples of embedded bit-banged communication protocols are the J1850, CAN and 1553. The J1850 is an automotive communication standard used in land-based vehicles supporting data rates such as 100 kbps. CAN is a similar multi-master protocol which is very widely used. Almost every passenger car currently manufactured in Europe features at least one CAN network. The MIL-STD-1553 is an interconnect standard adopted by the U.S. Department of Defense for use in the military and their contractors, primarily for avionics in aerospace equipment. 1553 features Time Division Multiple Access (TDMA) as the bus access protocol. Other examples of bit-banged pro-

tocols are infrared protocols for remote controls, software UARTs, modems and protocols such as I2C and RDS, a radio-based data broadcast service.

2.3 ASTI

Traditional STI methods provide *synchronous* thread progress within integrated functions. The real-time guest thread must have enough coarse-grain idle time (longer than two context-switches) to allow the remaining non-integrated portions of the host threads to run and avoid starvation. Software-implemented protocol controllers suffer significantly from this problem. Typically, there are two distinct threads of control in bit-banged implementations. There is a communication or primary (real-time guest) thread of control working alongside of other secondary (host) threads. The fine-grain idle time that is present in the primary thread is too short to be used for other threads, as it does not allow the context switches needed to switch between the primary and secondary threads. The primary thread "bit-bangs" each bit of the protocol onto or off of the bus, leaving only sub-bit duration fragments of idle time. This fragmented time is typically too short to recover through context switching, so only the primary thread can execute during message transmission or reception. Heavy network utilization can starve the secondary threads, leading to poor or unacceptable system performance.

The fine-grain idle time is imposed upon the primary thread due to the bit-rate of the protocol. ASTI aims to reclaim this fine-grain idle time. The primary thread is usually structured as three layers. The lowest layer is the bit-level layer which interacts directly with the physical bus. ASTI currently attempts to reclaim only the idle time that is present at this level. A sample distribution of this idle time is shown in *Fig.2.2.* The pattern shown repeats for each bit transmission or reception in an execution of the protocol.

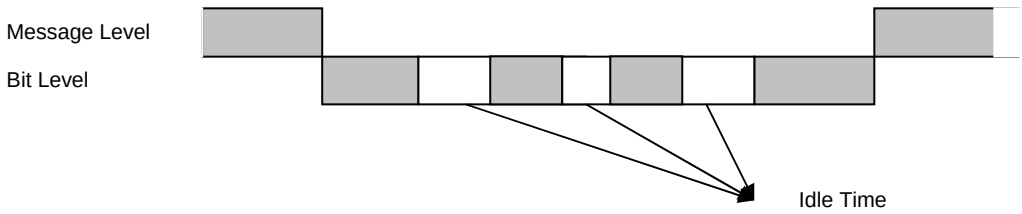


Fig.2.2 *Idle time in primary thread*

Idle time is reclaimed by allowing the secondary thread to execute in the idle time intervals. Current ASTI techniques perform certain transformations on both the primary and the secondary threads to reclaim the idle time. Transformations on the primary thread include

1) Padding the time between the message level function executions to the worst case interval.
2) Accumulating idle intervals adjacent to each other by removing the intervening guest code and building larger idle intervals *(Figure 2.3)*.
3) Inserting cocalls at the start of each idle interval in the bit-level function.

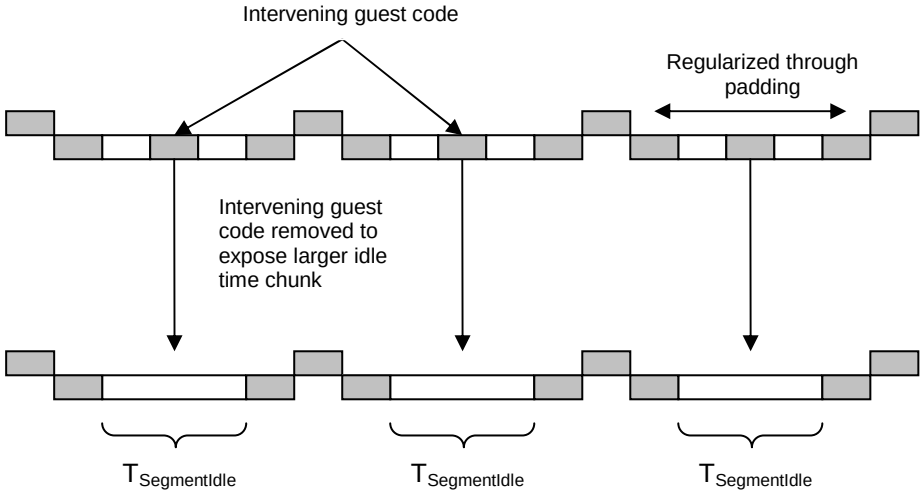


Fig.2.3 *Building larger idle chunks in the primary thread*

The secondary thread code is made to execute in this expanded idle interval in the primary thread. This is done by transforming the primary and the secondary threads to be structured as co-routines. Cocalls are inserted at appropriate places in the primary and secondary threads to transfer control back and forth between the threads. In the case of the primary thread, cocalls are inserted at the start of each idle interval.

In the case of secondary thread, cocalls transferring control back into the primary thread are inserted at a constant interval of $T_{SegmentIdle}$. The intervening guest code that was removed from the primary thread is integrated into the secondary thread code at the appropriate locations *(Figure 2.4)*.

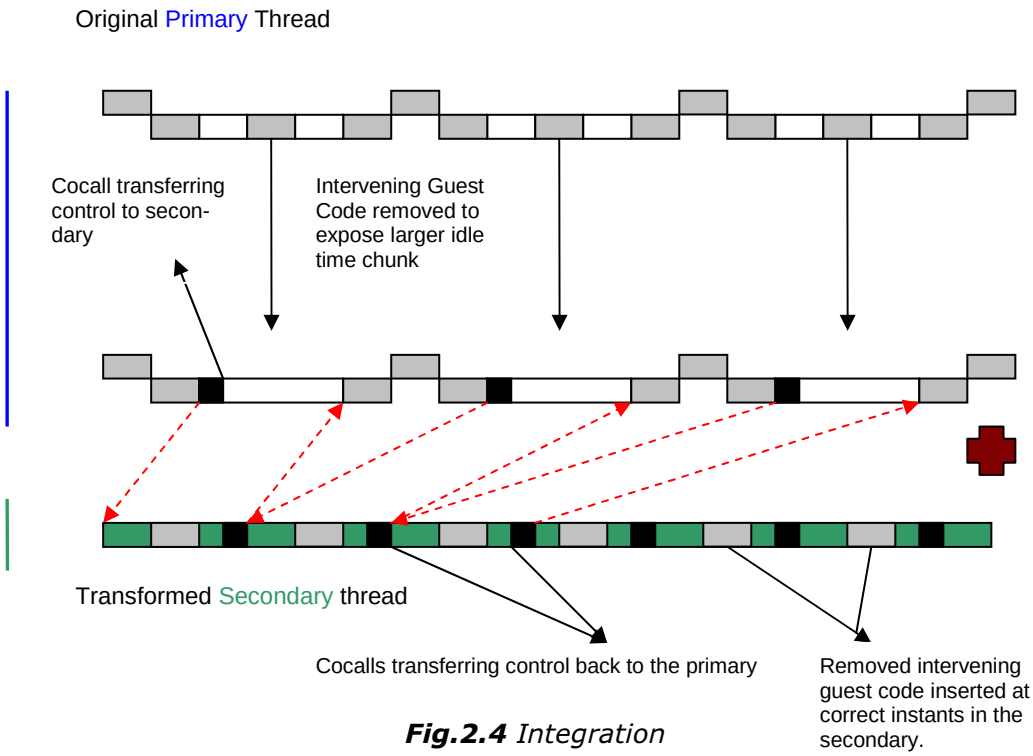


Fig.2.4 *Integration*

There can be multiple primary threads in a BBCP implementation (such as a data reception thread and a data transmission thread). Each primary thread will have its own distribution of idle time and the intervening guest code that is removed will be different for each thread. Frequently, control may need to switch in the middle of a thread's execution, to the other primary thread. For example, if there is data waiting to be received

while data is being transmitted, then control will need to switch to the data reception thread. Therefore, additional copies of the secondary thread with corresponding transformations for each primary thread are normally needed. This can be mitigated by using guarded cocalls and guarded guest code, described in [Kum02], which allow a single copy of the secondary thread to be used for all primary threads. The two secondary thread transformations described above, can then be incorporated into a single copy of the secondary thread, for all the primary threads.

Chapter 3

Related Research

3.1 Procedure Inlining

Procedure inlining has been a well-known optimization technique for over three decades [AC72]. Procedure inlining works by replacing a procedure call with the actual body of the procedure. It has traditionally been used to trade off code size for execution speed. Inlining may lead to three kinds of benefits [AFSS00]. First, there could be a gain in execution speed due to direct effects such as the reduction in the number of call and return operations. In low-end RISC processors procedure calls are one of the most expensive source language operations [PA82]. This may not be true in high-end processors which can have hardware support like register windows to minimize register saves and allow parameter passing, during procedure calls. Secondly, there could be benefits due to indirect effects such as cache, virtual memory. Thirdly the compiler could use data flow properties of the calling context to specialize the inlined body or optimize the caller due to data flow properties in the inlined procedure. Inlining thus increases the potential for whole program optimization. Several code improving techniques may be applicable after a procedure has been inlined [RG89, CMCH92]. These include register allocation, code scheduling, common sub-expression elimination, constant propagation and dead code elimination. Defining subtasks as procedures and then invoking the procedures appropriately to co-ordinate and accomplish the original task significantly improves ease and speed of software development. Therefore structured programming for software development is highly preferable. Inline expansion will allow a reduction in a program's development complexity while also allowing a reduction of the overhead in invoking procedures during execution.

Unfortunately, inlining can be accompanied by potential costs. The possible negative effects of inlining include - increase in code size, increase in the number of I-cache

misses, increase in register pressure leading to more spill and fill code, increase in stack pressure with excessive inlining . Inlining can also significantly increase the compile time of programs.

Compilers generally rely on heuristics to find the best tradeoff among these benefits and negative effects. The heuristic is basically an algorithm for selecting call sites to inline. Inlining compilers can be classified into two cases based on the kind of heuristics used – those that use *static inlining heuristics* and those that use *dynamic inlining heuristics*. Static heuristics rely on just the program text to guide the inlining decision without using any runtime information. Dynamic heuristics rely on the runtime profile information of the program to guide the inlining decision.

Inlining algorithms are generally modeled after the knapsack problem. Formally, the knapsack problem is the problem of optimization under a constraint - given a knapsack of a certain capacity and some items each occupying a certain amount of space and of a certain value, what is the best combination of items that one can choose, such that the items can fit into the knapsack and be of collective maximum value. With respect to the inlining heuristics, the individual items are the candidate procedures for inlining. Inlining a candidate procedure will lead to some benefit (value) at some cost (code expansion etc.). Therefore the heuristics attempt to choose a subset of the candidates such that the maximum benefit is obtained while not exceeding some cost constraint.

The benefit is usually modeled as the savings due to the avoidance of run-time procedure linkage (procedure epilogue and prologue), potential for optimization and others. The cost is usually the resultant code expansion, increase in stack pressure and other resource conflicts. In dynamic heuristics, the benefit is more accurately modeled as the minimization of the number of run-time calls to a procedure. In static heuristics, a distinction cannot be made between different call sites to the same procedure and therefore an *"all-or-none"* policy is followed - i.e. all static calls to a procedure are treated the same. It is not always possible to accurately measure the benefits and negative effects that are caused by inlining one particular call site. Therefore some measures which are

strong indicators of the benefit/cost potential of an inlining operation are used as an estimate. These are usually - execution time saved and code expansion, respectively.

Procedure inlining is not very clearly defined when it comes to recursive procedures. Unbounded inlining will lead to infinite code growth. Therefore in the case of simple direct recursion, usually inlining is performed up to some limit of *n* levels.

Inlining a procedure may result in the creation of many versions of the procedure. For example, consider a procedure *P* which has some call sites within its body. Supposing that the calls within *P* were inlined, then we have two versions of the procedure, *P* and *P'* which are semantically equivalent. When a subsequent decision has to be made to inline a call to *P*, either one of the equivalent definitions can be used, the benefits and disadvantages of which would vary accordingly. Formally there is a choice between inlining the original, the current or any version of the procedure. The actual choice made will impact the effectiveness of the inlining done. Intuitively, in inlining work, the current version of a procedure is always used. In [KR98], it is shown that this might not always be the best strategy. Flexibility of a strategy is defined to be the set of possible inlining plans that are arrived with a particular strategy and power of a strategy is defined to be the most optimal plan that can be arrived with a particular strategy. While intuitively any version inlining is the most flexible policy, it has shown that original version inlining is at-least as flexible as any version inlining and is more powerful than current version inlining.

3.2 Procedure Cloning

Procedure cloning is an inter-procedural optimization technique that works by creating specialized copies of a procedure. To clone a procedure, the compiler replicates the procedure and divides the incoming calls to it between the original and the created clones. Partitioning the set of clones and mapping the calls suitably can lead directly to more optimization or lead to an increase in the optimization potential. Procedures impose a barrier on the compiler's potential regions of optimization. Procedure cloning can be

used to expose larger regions of optimization. Furthermore, compilers, traditionally, have believed that a procedure should have only one implementation even though it might be called in a variety of data flow and control flow contexts [CHK89]. Specifically, compilers can apply procedure cloning to implement multiple versions of each procedure, each tailored to suit some particular data flow or control flow path. Procedure cloning can be used as an alternative to inlining, as there can be comparatively less code expansion with cloning.

Procedure cloning has been applied in [CHK89] to the *matrix300* program from release one of the SPEC benchmarks. Excessive inlining is needed to apply a greatly beneficial *unroll and jam* optimization on the benchmark's implicit nested loop structures. Procedure cloning is used to reduce the amount of inlining needed by doing a forward interprocedural data flow analysis to gather information on the parameters passed to each nested procedure. Based on this information, equivalent sets of calls are merged and assigned to corresponding clones. Then a final inlining phase is executed, transforming the code structure such that, the desired optimizations can now be performed. Selective specialization, discussed in [DCG95], corresponds to using procedure cloning to reduce dynamic dispatch overhead in object oriented languages. Procedure cloning has also been applied in [SD03] along with STI to create clones that have procedure level parallelism converted to instruction level parallelism.

3.3 Call Graphs

Call graphs are the most commonly used models of control flow between procedures of a program. A call graph ***G(V,E)*** is a graph with *N* nodes representing the procedures in the program and *E* edges connecting procedures and representing *caller-callee* relation. Specifically an edge from node *A* to node *B* indicates that there is a call site in procedure *A* to procedure *B*. Since there may be multiple such call sites in a single procedure, each edge is labeled (usually) with the *address* or *label* of the actual call instruction. The edges maybe additionally labeled with a cost function which indicates the cost of an inlining operation on that edge. The call graph is *topologically sorted*, i.e. for all edges

(u, v) in the graph, node *u* appears before node *v* in the topological order. *Figure 3.1* shows a sample call graph.

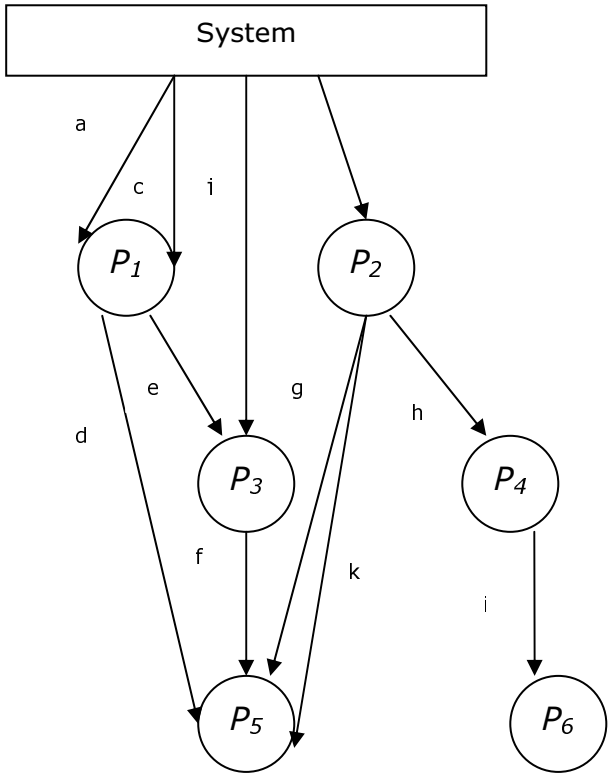


Fig.3.1. *Sample call graph*

Call graphs maybe *static*, *dynamic* or a *hybrid* of both. A static call graph represents the static calling relation between procedures. Therefore even though a call may never occur during the actual execution of a program, there will be an edge representing the call. A dynamic call graph, rather, has only edges representing calls that were encountered during an actual execution of the program. This information maybe obtained through profiling tools and techniques. An important difference between static and dynamic call graphs is that in static call graphs it is not possible to distinguish easily between two different edges between the same pair of nodes, whereas in dynamic call graphs two different edges between the same pair of nodes could have different assigned weights based on their frequency of execution. Static call graphs can use some properties of the loca-

tion of a call site to determine its inlining benefit relative to another call site. For example, inlining a call site which is held within a loop is intuitively advantageous compared to a call site that is outside of it. An inlining operation gets reflected in the call graph by the removal of the corresponding edge. Node weights (code size) get updated correspondingly. If a node loses all its incoming edges it basically represents dead code and can be eliminated from the program. Call graphs reduce optimizations which work with control flow, to well known graph theoretic problems.

Chapter 4

Contributions: New Methods

4.1 Applying Procedure Inlining and Cloning

Procedure inlining is a viable technique to remove procedure calls in the secondary thread. Inlined procedures do not have the timing variability problem with respect to cocalls and intervening guest code. This is because each call site gets its own body of the procedure replacing the call. Procedure inlining though, leads to code expansion. The effects of procedure inlining might be minimal or drastic, depending upon the size of the procedure and the number of calls to it that need to be inlined. For example, deciding to inline a procedure of size *m* bytes with a total of *N* incoming calls, will lead to a code expansion of *(N-1)*m*. When a call site is inlined, the body of the procedure in which the call site occurs, can now be of two different versions. The code expansion effect discussed above is the case when always using the original version of the procedure to inline. The effect of code expansion is more complex when there is procedure nesting involved and when any version of the procedure can be used to inline a call site.

A straightforward solution to the control flow problem with ASTI would be to replace all call sites with the body of the target procedure. While this solution maybe feasible for simple and small secondary threads, it can quite easily cause an overflow of the target micro-controller's available flash memory. For example, one of the benchmark secondary threads has a base code size of *5216 bytes*. When this secondary thread is fully flattened by inlining all the procedures that occur in it, the code size grows to *13176* bytes. This can require the next higher micro-controller configuration to accommodate all of the code. It must also be noted that there will be other software threads that might be vying for flash memory space.

There is usually a very hard limit on code size in most embedded systems. *Table 4.1* shows per unit cost of three micro-controllers that have *comparable* configurations except for the available flash memory. Excessive inlining can lead to code size for the secondary thread exceeding the available flash memory for a particular configuration, and hence requiring the next higher configuration with more flash memory. In the case of embedded systems that have a large potential market, this raise in per unit cost of the micro-controller may be intolerable. Inlining also leads to increased register and stack pressure.

Table 4.1 *Microcontroller configurations and high volume pricing*

Chip	Flash Memory Size in kilobytes	Clock Speed in MHz	Cost per unit at 50k Units (in dollars)
ATMega8L	8	0-8	2.35
ATMega161	16	0-8	5.10
ATMega323	32	0-8	7

The alternative to inlining a procedure is to remove the timing variability by transforming all calls to the procedure so that they occur at a fixed interval from the last cocall. Then the procedure body can undergo the secondary thread transformations making suitable assumptions about the starting time. Removing timing variability can be done by adding padding code, which can be *nop loops* or *delay loops*, to the body of the caller. The problem with padding code is that executing such code wastes processor cycles and brings down the processor efficiency which STI primarily aims to increase.

Procedure cloning can be used as an extension to the alternative of inlining. The time intervals between calls to a procedure and the last occurring cocall could follow a certain clustering pattern. A clone of the procedure could be created for each such cluster point. This can greatly reduce the amount of padding cycles added statically and by extension provide significant savings during run-time. By carefully partitioning the calls to the procedure among the clones, a clone can inherit an environment in which the secondary thread transformations can be performed in the most processor-efficient way. Suitable heuristics and an algorithm are needed to reconcile between inlining and cloning

based on code expansion and processor efficiency. The proposed solution employing inlining and cloning suitably is described in the next section.

4.2 Approach for Relaxing Constraints

The approach for relaxing control flow constraints involves two phases. In the first phase, a set of call sites will be chosen to be removed through procedure inlining. This is because, typically, there will be some amount of code expansion that can be tolerated by the system at hand. This enables us to use procedure inlining to remove some procedure calls. Call sites which are not inlined have to be cloned and handled, with potential insertion of padding cycles which bring down processor efficiency. Therefore, for a given code size constraint, the initial phase will have to remove as many call sites as possible through procedure inlining.

The second phase will attempt to transform the remaining procedures and calls so that cocalls can be inserted in the usual fashion. Procedure cloning will be used to do this in the most optimal fashion.

Outline of approach:
Algorithm: Relaxing control flow constraints
Input **:** Original secondary thread with procedures and procedure calls
Output **:** Transformed secondary thread with the control flow regularized.
Method:

(1) Form a call graph of secondary thread.
(2) *(Phase one)* From the set *S* of procedures in the secondary thread, select a subset *I* of procedures to be inlined, such that a maximal number of call sites are removed while not exceeding some code expansion limit *L*.
(3) Inline the set of procedures in *I*.
(4) *(Phase two)* Transform the set *S-I* of procedures remaining in the secondary thread to remove timing variability. Use procedure cloning to reduce the amount of overhead introduced due to the transformations.
(5) *(Phase three)* Now that the secondary thread has been fully regularized, cocalls can be inserted in a straightforward fashion at regular intervals of $T_{SegmentIdle}$ [Kum02].

The inlining operations in phase one will increase the potential for certain optimizations that could previously have been performed only inter-procedurally. These are mainly

traditional optimizations that can be performed after procedure inlining. One such optimization that, intuitively, can produce significant benefits is partial redundancy elimination (PRE) [MR79]. Partial redundancy elimination combines and extends *common subexpression elimination* and *loop invariant code-motion*. An expression is said to be partially redundant at point *p*, if it is redundant along some, but not all paths leading to *p*. PRE converts these partial redundancies into full redundancies by moving code to a suitable point and then removes the redundant expressions. Such optimizations are left as future work in an intermediate phase between phases one and two. Thus code optimization can be an added bonus due to the inlining in phase one.

4.3 Phase One

4.3.1 Description

As described earlier, phase one aims to arrive at an inlining plan which reduces as many call sites as possible while satisfying a code expansion limit. From the point of view of the inlining decision, all call sites have the same *relative* benefit. Such a formulation reduces the problem to the easy 0/1 knapsack problem [GJ79], for which a linear time solution exists. In this case the optimal solution is to sort the items in order of increasing cost and insert them into the knapsack in this order until nothing fits. But such an easy solution cannot be adopted because of the problem of mutual dependence of call sites. Choosing to inline a particular call site may produce new call sites, which essentially changes the number of items in the knapsack formulation. Also while inlining call sites which are nested, the code size (cost) of the outer procedure will grow. These issues are not addressed in traditional knapsack formulations.

Procedure inlining under a code expansion constraint for embedded systems is discussed in [LM99]. The core of the solution is a branch and bound algorithm for minimizing the number of dynamic calls while satisfying a code expansion constraint using procedure inlining. The algorithm allows an exploration of the entire solution space in a *comparatively* shorter time. Branch and bound algorithms work by starting at the root problem and applying known lower and upper bounds. If the problem exceeds the bounds, then it is cut off from the search space. If the solution space satisfies either of the bounds the

solution space is partitioned into two and the two spaces are recursively examined in the same way. The technique presented minimizes the number of *dynamic* procedure calls. The primary concern for STI is not the optimization of dynamic calls, but rather the number of *static* call sites.

4.3.2 Proposed technique

The technique presented in [LM99] is adapted and modified for use in the inlining phase of the constraint relaxation process. The algorithm arrives at a Boolean inlining decision for each procedure – either the procedure is inlined or it is not. If each procedure forms a bit in a bit vector, then the algorithm attempts to assign values to each bit such that the total number of static call sites is minimized.

For a given C source code, the following information is first gathered.

- M, the subset of all the procedures S which are candidates for inlining.
- The code size for each procedure $B(f_i)$.
- The number of static calls that occur in the code for each procedure $N(f_i)$.
- An absolute code expansion limit L.

The same bit-vector technique that is presented in [LM99] to represent the set of functions that are to be inlined is followed here. Formally, IV $(b_1,b_2,b_{3\ldots}b_n)$ is the inlining vector. A value of one (zero) for any b_i indicates that all calls to the corresponding procedure f_i are (not) to be inlined.

At any instant, the total number of static system calls is given by,

$$N(IV) = \sum_{fi:bi=0} N(f_i) \tag{4.1}$$

The theoretical minimum value is 0, when IV = (1,1,…1). The total code size at any time during the algorithm is given by,

$$S(IV) = \sum_{i=0}^{M} S(f_i) \tag{4.2}$$

$$S(f_i) = B(f_i) + \sum_{f_j : b_j = 1} C(f_i, f_j).S(f_j)$$

This is a recursive definition that can be applied in the topological order of procedures - in the call graph. The recursion will terminate at the leaf node. The algorithm proceeds on the basis of the following analysis.

Let $IV=(b_1,b_2,..,b_{i-1},x,x\ldots)$ be the current inlining vector, where the values for b_i to b_{i-1} have already been fixed and we need to evaluate values for the bits from position *i* through *n*. N_{min} is the minimum value of *N(IV)* found so far. Initially N_{min} is set to be equal to the original number of static calls appearing in the code.

There are four possible cases now.

1) No more *x*'s appear in the *IV*. Therefore total code expansion S*(IV)* due to this *IV* is calculated and if it is less than the limit *L*, N_{min} is set to *N(IV)*, if the current N_{min} is greater than *N(IV)*.
2) Otherwise we check if f_i can be inlined without violating the code size constraint by temporarily setting b_i to 1. A lower bound for this case is obtained by setting all the remaining *x*'s to 0. Therefore *IV* is set to $(b_1,b_2,..,b_{i-1},1,0,\ldots 0)$. If *S(IV)* is greater than *L*, then this part of the search space can be pruned away.
3) If the pruning in step 2 cannot be done, we check if a new minimum *N* value is possible by not inlining f_i. This can be done by setting b_i to 0 and all the remaining *x*'s to 1. The corresponding *IV* is $(b_1,b_2,..,b_{i-1},0,1\ldots 1)$. Now, if *N(IV)* is greater than N_{min} then, not inlining f_i cannot improve the current best solution. Therefore the search space for b_i=0 can be excluded from analysis. Therefore b_i is set to 1 and the search continues through the remaining bits.
4) In the worst case, neither of the two pruning steps could be possible and hence both the cases (b_i=0 and 1) must be recursively evaluated and the best solution from among the two chosen.

The worst case complexity of this algorithm is exponential in *N*. But in many cases the pruning in steps *2)* and *3)* can be applied. This allows the algorithm to execute in a rela-

tively shorter number of steps. The algorithm is outlined in the next section. The way that the algorithm is applied is by tuning the values of the code size limit and choosing the appropriate inlining plan based on the tradeoff between the expected code expansion and the amount of reduction in call sites that are obtained. The code size limit can be specified as a percentage of the original code size.

4.3.3. Phase One Algorithm

Algorithm: Finding the most optimal inlining vector - Min_IV

Input : Inline vector $IV(b_1,b_2,b_3..b_n) \varepsilon (0,1,x)$.
Output : Inline vector $IV(b_1,b_2,b_3..b_n) \varepsilon (0,1)$.
Method :

```
(1)   if there are no 'x' bits in IV then
(2)     if S(IV) < L and N(IV) < Nmin then
(3)       Nmin := N(IV)
(4)     endif
(5)     return IV;
(6)   endif
(7)   Let i be the first index in IV such that bi = 'x'.
(8)   if S((b1,b2..bi-1,1,0,0..0)) > L then
(9)     return Min_IV((b1,b2..bi-1,0,x,x..x))
(10)  else if N((b1,b2..bi-1,0,1,1..1)) > Nmin then
(11)    return Min_IV((b1,b2..bi-1,1,x,x..x))
(12)  else
(13)     IV0 := Min_IV((b1,b2..bi-1,0,x,x..x))
(14)     N0  :=  N(IV0)
(15)     IV1 := Min_IV((b1,b2..bi-1,1,x,x..x))
(16)     N1  :=  N(IV1)
(17)     if N0 < N1 then
(18)       return IV0
(19)     else
(20)       return IV1
(21)     endif
(22)  endif
```

4.3.4 Observations about the algorithm

- Either all call sites of a procedure are inlined or not. Call sites are not examined individually. This is an acceptable simplification of the algorithm. Allocating a bit to represent the inlining decision for each call site in the vector, while feasible will bloat the vector to many times its original size and will significantly increase the execution time of the algorithm.
- Traditional approximation algorithms for the Knapsack problem cannot be used because of the mutual interdependence of call sites on one other as described earlier.
- The algorithm in [LM99] aims to minimize *dynamic* call frequency. This value is independent of changes in the inlining vector. The modified algorithm for STI aims to minimize the number of *static* call sites. Based on the current configuration of the *IV*, new static call sites could be created. Hence changes in *IV* will have a global effect on not only *S(IV)* but also on *N(IV).* This is slightly different from the approach in [LM99] where the total dynamic call frequency of a function remains unchanged with changes in the *IV*.
- [LM99] contends that this kind of a branch and bound solution is a good idea compared to faster approximation algorithms because even minimal changes in the code size limit causes large and global changes in the IV, rather than small, progressive changes. Therefore, examining the full solution space using this branch and bound technique is a good idea.

4.4 Phase Two

4.4.1 Outline of Phase Two

Phase two handles all the call sites that remain after phase one is complete. Any procedure can be transformed so that the interval between its starting times and the last occurring cocall is regularized to a single value among the distribution. The number of cycles left before a cocall must be inserted, at the entry point of a procedure, can be defined as the *phase* (Δc) of the procedure with respect to that call site. The worst case phase is de-

fined to be the least value of Δc. Regularizing the phase at each call site can be done by applying padding code at the call site so that the phase is padded to the worst case phase. This resolves possible conflicting cocall placements. This is illustrated in *Figure 4.1*.

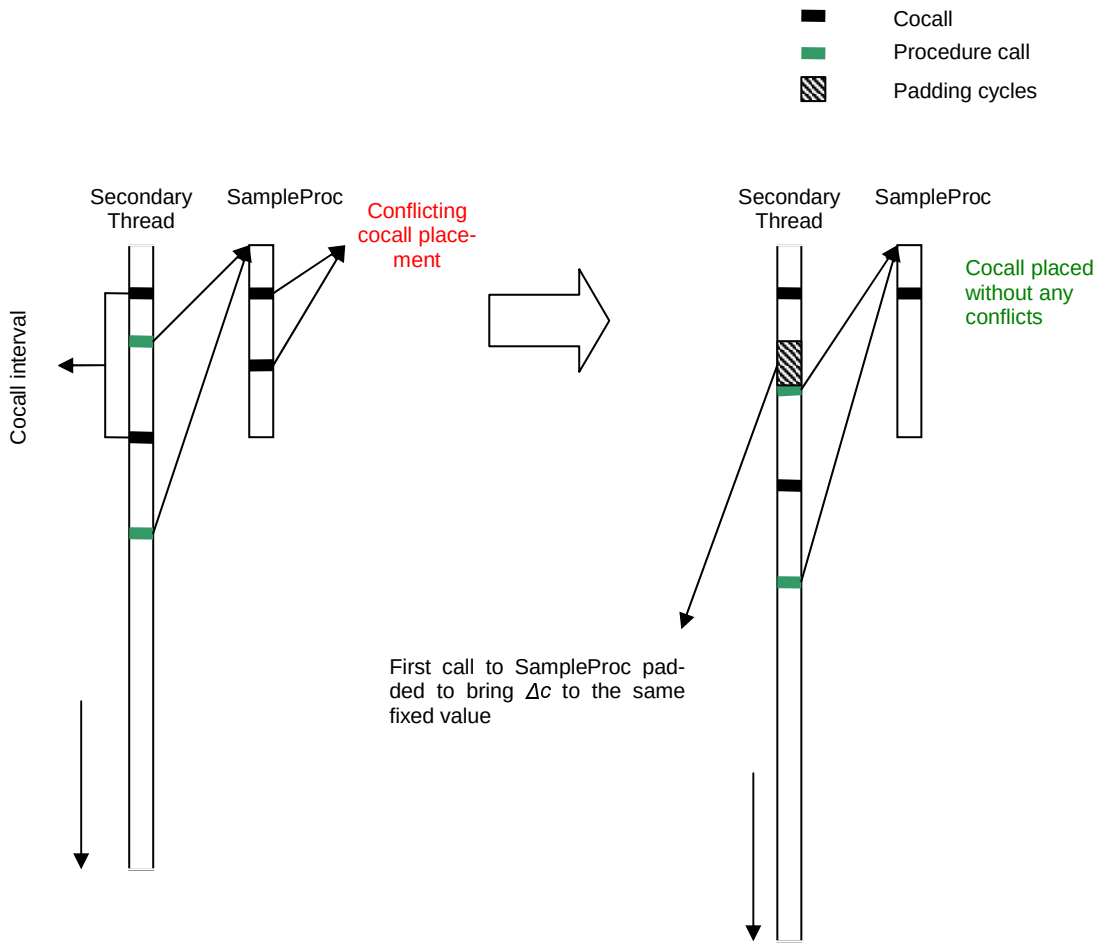


Fig.4.1 *Resolving conflicts*

Regularizing call sites without inlining them has the advantage that there is no code expansion. But padding code typically consists of *nops* or, as an optimization, delay loops. Such code represents no useful work and brings down the processor efficiency.

Procedure cloning can be used to reduce the amount of padding code that needs to be inserted. For each procedure, the phase at each call to the procedure is plotted and its

distribution analyzed. The clustering pattern of Δc might justify partitioning the set of calls and forming a clone for each set. This is illustrated in *Figure 4.2.*

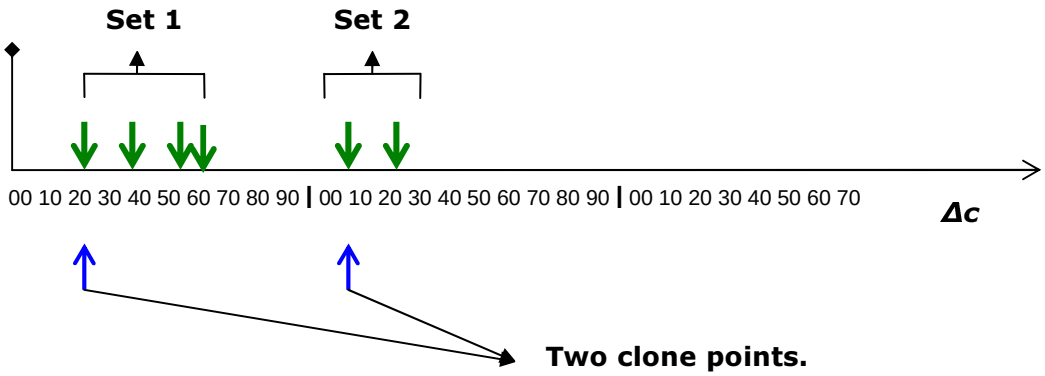


Fig.4.2. *Plot of cycles left to insert next cocall for a sample procedure*

In the example in *Figure 4.2*, if no cloning had been performed, the total padding would be roughly to the order of 275 cycles. This is because all the call sites would have to be padded to the Δc which is at 20 cycles. Creating one more clone as illustrated would bring down the total amount padding to around 105 cycles. This can be reduced further, if a third clone is formed, to 45 cycles. Unfortunately, producing three clones would inflate the code size of the size of the procedure under consideration by a factor of three.

The amount of clones generated for each procedure can be controlled by the *Clone Ratio (CR)*. For each procedure its clone ratio can be defined as in *Equation 4.3*.

$$CR = N_c/N_{cl} \tag{4.3}$$

Where,
CR - Clone Ratio,
N_c - Number of clones created,
N_{cl} - Number of incoming calls to the procedure.

Thus the clone ratio can vary from a maximum of 1, which indicates that there is a clone for each call site to the procedure, to $1/N_{cl}$ – which indicates that there is a single copy of the procedure to handle all the incoming calls. Intuitively, a clone ratio of one will mean

that no padding cycles need to be generated, while the minimum cloning ratio for a procedure will result in the maximum amount of padding cycles generated. The effects of the clone ratio on the padding cycles produced and the resultant code expansion depend on the Δc distribution and the size of the procedure. Since it is not convenient to algorithmically determine the best tradeoff between clones created (code size) and the reduction in padding cycles obtained for a particular clone ratio, the clone ratio can be allowed to be tuned for each procedure during phase two. Therefore by examining the results for a variety of clone ratios, the best values can be chosen and the corresponding cloning operation performed.

4.4.2 Partitioning Call Sites Efficiently

Efficiently clustering the incoming calls to a procedure so as to reduce padding code basically involves partitioning the data set represented by all the phases of a procedure, such that phases that are close to each other go into the same data set. Each set will have a representative phase to which the rest of the set's phases get padded to.

Clustering algorithms tend to be partitional or agglomerative. In partitional strategies, all the elements are in a single data set initially and the set is partitioned recursively till the required number of sets is generated. Agglomerative approaches start with each element being in its own set and then recursively merging sets till the number of sets is brought down to the required value.

Clustering algorithms use different metrics to gauge the similarity between two elements of a set (otherwise known as the distance between the sets). The metrics usually fall into three categories – Euclidean metrics, non-Euclidean metrics and semi-metrics. Of these the Euclidean metrics are the simplest metrics which calculate the straight line distance between two points in Euclidean space. The other two metrics are more complex ways of calculating the distance between two points, where a straight-line distance cannot be measured or is not applicable. Clustering algorithms also vary in the methods by which they apply the distance metric between sets. For example, two sets could be chosen for

fusion based on the average distance between the centroids (a representative average value) of each set. This is known as average linkage clustering. In single linkage clustering, the distance between two sets is defined to be the minimum distance between any two elements in each set. Similarly there could be complete linkage clustering, wherein the distance between two sets is the distance between the two furthest points in each set.

Phase two uses a hierarchical clustering algorithm known as minimal spanning tree (MST) clustering algorithm, with a Euclidean metric to evaluate the distance between any two points and *within group linkage.* The distance between two phases, Δc_1 and Δc_2, is given by *Equation 4.4*.

$$d = MIN(d_{straight}, d_{circular}) \tag{4.4}$$

Where,

$$d_{straight} = ABS(\Delta c_1 - \Delta c_2) \text{ and} \tag{4.5}$$

$$d_{circular} = \Delta c_1 + (\Delta c_d - \Delta c_2), \text{ if } \Delta c_2 > \Delta c_1 \tag{4.6}$$

$$\text{else } \Delta c_2 + (\Delta c_d - \Delta c_1), \text{ if } \Delta c_1 >= \Delta c_2$$

Δc_d is the duration of a cocall ($T_{SegmentIdle}$).

Consider the example in *Figure 4.3* – The perimeter of the circle can be thought of as Δc_d, the cocall duration. The length of the acute arc is $d_{straight}$, while the length of the obtuse arc is $d_{circular}$. When $d_{circular}$ is the minimum distance, then a cocall needs to be inserted at the appropriate place within the padding code when the Δc value goes to zero.

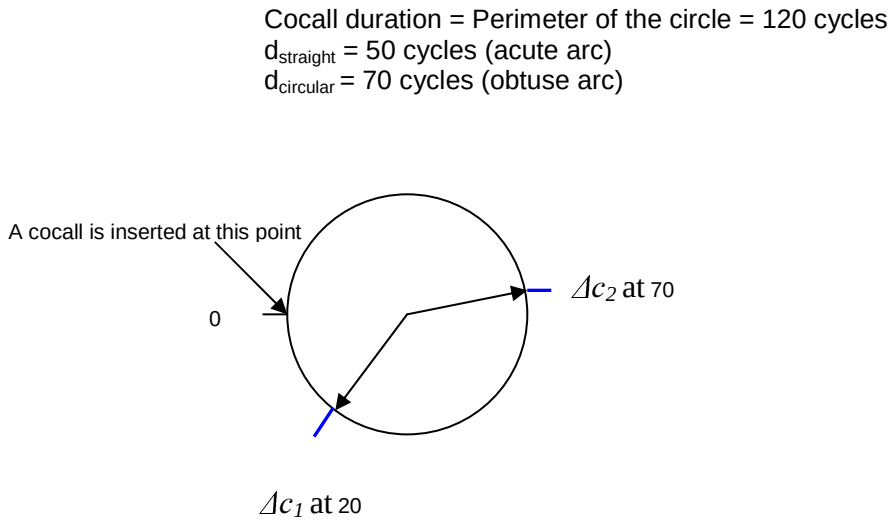


Fig.4.3. *Distance between two phases* Δc_1 and Δc_2

Within group linkage is the variance of the distance of all the elements in the set from a single representative element. The MST algorithm takes an agglomerative approach, where in each merge step, it combines the two sets that require the *minimum amount of padding* if a single clone is to handle all the calls in both the sets. The algorithm can be informally described as follows.

Initially all the elements of the distribution are in a set of their own. In each merge step, the algorithm picks the two sets that have the minimum resultant within group linkage. In the context of padding code, within group linkage is defined to be the amount of padding code that would need to be added if all the calls in the set are mapped to a single clone. The algorithm then merges these two sets. This merge phase continues till the number of sets is brought down to the required number as controlled by the clone ratio for the procedure. The merge phase is comparable to forming an edge between the two sets being merged. This edge is also the minimal edge at that step. Therefore the forest that is formed is a minimal spanning forest. A sample run of the algorithm is illustrated in *Figure 4.4*, for a cocall duration of 300.

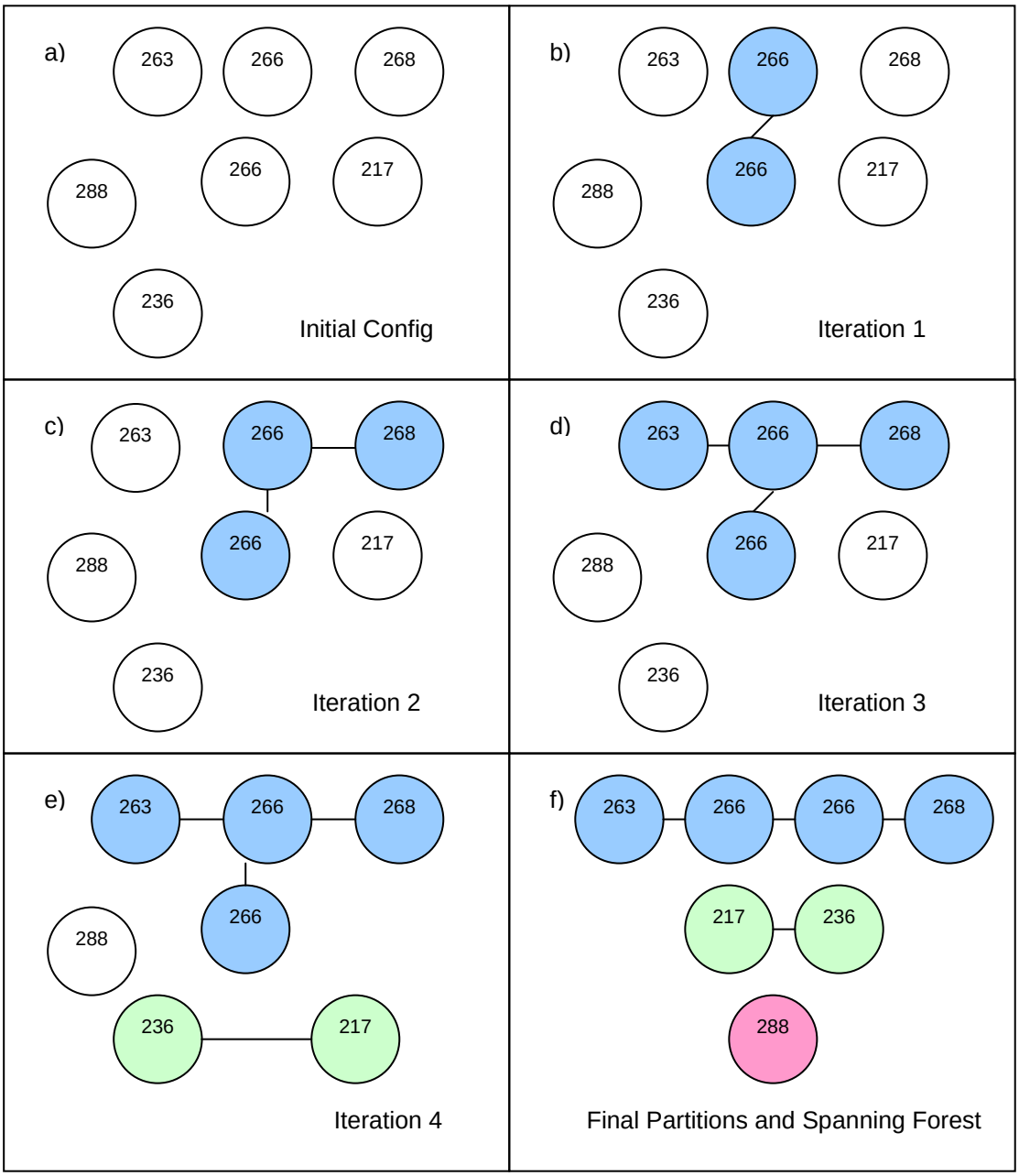


Fig.4.4. *Illustration of minimal spanning tree clustering algorithm*

The sample run uses a clone ratio of 0.5. Without cloning, the padding cycles required would have been 285 cycles. When up to three clones can be formed, the amount of padding required drops to 30 cycles.

Algorithm: Minimal spanning tree clustering

Input : A(S_1, S_2, .. S_n), where S_i is set *i* containing the i^{th} incoming call to procedure *P*, which has *n* incoming calls.

Output : A(S_1, S_2, ..S_m), where S_i is the set of calls that must be retargeted to the i^{th} clone of the procedure and where *m* = CEIL(*CR***n*), the desired number of clones. *CR* is the desired clone ratio.

Method:

(1) Form an *n x n* table *Link*, where *Link[i,j]* contains the *within group linkage* of the resultant set when set *i* and set *j* are merged. *Link[i,i]* contains the within group linkage of set *i*.

(2) While number of sets in current iteration *j*, n_j, is not equal to *m*, the desired number of partitions, do

a. For *i, j, i* <> *j*, Pick an *i* and a *j* such that *Link[i,j]* is minimum.
b. Merge sets *i* and *j* into set *i*. Update column *i* of table *Link* with new within group linkage. Set all entries in column *j* of *Link* to ∞.
c. Delete set (row) *j*.
d. Repeat (2).

4.4.3 Intervening Guest Code

As explained before, in each $T_{SegmentIdle}$ of the secondary thread's execution, guest code removed from the primary thread to expose larger idle intervals, must be placed at appropriate locations in the secondary thread, so that the real-time work starts at the right time. This intervening guest code always starts at a fixed offset from the start of the idle interval. Therefore, placing the intervening guest code into the body of procedures is straightforward and follows the placement of cocalls in the procedure body, transferring control back to the primary thread. Once all call sites to a procedure have been cloned and handled as described in previous sections, placement of cocalls and intervening guest code can be done in the same manner as described in [Kum02] since the start time of each call to some clone of the procedure, relative to the last cocall, has now been fixed by the previous clustering pass.

4.4.4 Call Graph Processing Order

The entire call graph could be processed in different ways, removing the constraints on each call site in the processing order. Timing information (static start time) of each call node needs to be completely available before the call site can be processed. Proceeding top-down through the call graph, descending through the levels is one possible way. Timing information (start times) is readily available only for the call sites at the first topological level of the call graph. This is because, start times for nested calls cannot be determined unless the transformations for the parent procedure are fully performed. Proceeding top-down then, is the logical thing to do.

The location of call sites relative to each other makes it very hard to readily obtain timing information. This is described below.

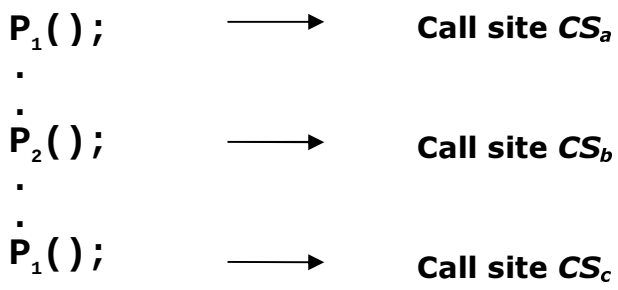


Fig.4.5. *Sample call site ordering*

Timing information for any call site will be available as long as the timing information for the procedure in which it is embedded is available. But calls to a procedure could be interleaved amongst calls to other procedures that are at the same topological level. In *Figure 4.5* a call to P_2 is sandwiched in between calls to P_1. This means that the timing information for call site CS_c would change depending on how call site CS_b and subsequently procedure P_2 are finally transformed. Therefore making assumptions about timing information for all calls to procedure P_1 and transforming P_1 might be invalidated by future transformations on other intervening procedure calls.

Two different approaches can be used to handle this.

1. The call graph could follow a partial ordering within each level, apart from the topologically ordering. In the example in *Figure 4.5*, two procedure clones could be formed for P_1 initially, say P_1 and P'_1 and the call graph could be rearranged to follow a left to right ordering such as (P_1, P_2, P'_1) within the same topological level. Therefore, call site CS_c would actually make a call to P'_1. Thus procedures that are to the right of a procedure P_i within the same level, occur after P_i in the control flow of the secondary thread. When proceeding through the call graph, procedures are handled left-right within a single topological level. This will ensure that when handling a procedure P_j, all procedures which can possibly influence this procedure's start time (all procedures which are at a topologically higher level or to the left of P_j in P_j's topological level), would have already been transformed and hence the start time can be determined without ambiguity.

2. Pad each procedure so that it lasts a multiple of the cocall duration and hence makes a cocall return exactly at the end of the procedure. In the example in *Figure 4.5*, this will let us handle both the calls to procedure P_1 (at CS_a and CS_c), even if the eventual transformations that procedure P_2 will undergo are not known at that point of time.

The latter option is much more appealing than the former. This is because the partial ordering technique will produce initial clones in an unpredictable fashion, for each procedure in the system. Intuitively, this could lead to drastic code expansion, producing an effect similar to inlining a majority of the calls to each procedure, although the actual effect depends on the calling and nesting patterns of procedures relative to one another. This initial cloning will also drastically reduce the potential for grouping clones based on Δc proximity. Padding all procedures to a multiple of the cocall duration is a reasonable simplification that provides a feasible solution to the processing of the call graph. However it is disadvantageous in the fact that it slows down the true progress of the secondary thread and results in reduced real-time performance.

4.4.4 Multiple Primary Threads

As explained before, there could be multiple primary threads which are to be integrated with a single secondary thread. For example, in the J1850 protocol implementation described in [Kum02], there are two primary threads – a bit transmission thread and a bit reception thread. Each of these threads has its own $T_{SegmentIdle}$ and intervening primary code. [Kum02] presents two solutions to the problem of multiple primary threads. The first solution uses a dedicated copy of the secondary thread for each version of the primary thread, with synchronization points to switch from one to the other. The second solution presents guarded cocalls and guarded intervening primary code. The guard is implemented using a predicate which tests certain bits in a dedicated register which indicate which primary thread's cocalls and intervening primary code need to be executed. Therefore based on which primary thread is currently executing, the corresponding cocalls and intervening code become enabled. Synchronization points are still needed when there is a switch from one primary thread to the other.

The strategy for cloning and handling procedures presented in earlier sections can be applied in a straightforward fashion if the first solution (using individual copies of the secondary thread) is adopted. The cloning and regularizing technique can be applied on each version of the secondary thread for the $T_{SegmentIdle}$ corresponding to the version of the primary thread to which this secondary thread is dedicated. This solution has the disadvantage that separate copies of the secondary thread lead to code expansion.

If guarded cocalls and intervening primary code are used, then the cloning algorithm needs to undergo some subtle changes. First the distance measure by which the similarity between two call sites is estimated should be changed from a simple Euclidean measure to one that incorporates distance in N-dimensions, where N is the number of versions of primary threads. Secondly, the padding code that is added before each call site, to bring Δc to a fixed value, should be guarded in the same way that cocalls and intervening guest code is guarded. Thirdly, the technique of padding each procedure to a multiple of a cocall, to simplify the call graph processing order, must be modified so that each procedure is padded to a multiple of each of the cocall durations corresponding

to the primary threads. This padding is also suitably guarded. With these changes a single copy of the secondary thread can be cloned and handled. However the analysis and transformations are much more complex than if they were to be done for a single version of the primary thread. The research presented here has not focused on multiple primary threads and leaves it to future work.

4.5 Implementation in Thrint

4.5.1 Overview

The constraint relaxation algorithm involving phases one and two has been implemented in the post pass compiler *Thrint*. The implementation starts with a regular structured secondary thread with procedures and procedure calls. At the end of phases one and two, the secondary thread code is flattened and regularized from the point of view of timing with respect to cocalls - i.e. it can be directly passed to the implementation done in [Kum02], which takes the primary and secondary threads and integrates them as coroutines. This is now possible because the thread would have been padded, at the end of phase two, in such a fashion so that all calls to a procedure occur at the same offset from the last cocall. Code expansion and wasted cycles are profiled during each phase of the algorithm. A breakdown of the manual and automated tasks is given in *Table 4.2*.

Table 4.2 *Breakdown of automatic and manual tasks*

Automatic	Manual	Description of Task
x		Form the call graph of secondary thread.
x		Topologically sort the call graph.
		Phase 1
x		Apply Min_IV algorithm for various code expansion limits.
	x	Choose an inlining plan.
	x	Inline procedures chosen in plan using *gcc*.
x		Profile code expansion and static call site reduction.

Table 4.2 *(Continued)*

		Phase 2
x		Pad jitter in each procedure's CDG.
	x	Pad blocking I/O loops to duration of a cocall.
	x	Calculating durations of each procedure in reverse topological Order.
x		Obtain timing information on Δc distribution for each proce dure.
x		Apply minimal spanning tree clustering algorithm for each procedure and get cloning plans.
	x	Choose desired cloning plan.
x		Profile padding cycles inserted and code expansion due to Clones.

4.5.2 Code Transformations

The only code transformation required in phase one is the inlining of procedures according to the arrived inlining plan. The inlining decision made in phase one, is executed by adding *inline* keywords or similar constructs to direct/force the compiler to do the inlining transformations. The *gcc* compiler with which the research was performed can be forced to inline calls to a procedure by using the modifier *extern inline* in the definition of the procedure. However a limitation with *gcc* is that inlining occurs only at optimization levels greater than zero, whereas, *Thrint* works only with optimization level zero, the reason being that *gcc* produces unstructured code at higher optimization levels. Future work aimed at techniques to convert unstructured constructs into structured ones will remove this constraint and inlining can be easily done using *gcc* itself. For this research, the inlining operation was performed manually. The resultant code proceeds to phase two.

Certain code transformations are needed in phase two, before the minimal spanning tree clustering algorithm can be run. First, the entire CDG is padded to eliminate jitter relative to a cocall. This is described in detail in [Kum02]. For example, predicates produce jitter in the execution path based on whether the branch is taken or not taken. Padding code is added to either of the branch such that the cycles left to insert a cocall again be-

comes equal irrespective of which branch was taken. Similarly loops are transformed such that the cycles left before a cocall must be inserted is equal irrespective of whether the loop is entered for the first time or in subsequent iterations. Every blocking I/O loop is made to execute for the duration of a single cocall in each iteration of the loop. These transformations eliminate timing variability in the CDG, after which the algorithm for phase two can proceed.

In phase two, when a call site is handled, the cocall insertion process is carried over into the body of the corresponding clone (if any) which is most efficient for this call site. Each clone makes some assumption about its starting time relative to the last cocall. Padding code is added at the call site, as needed, to make sure that the call to the clone occurs at the right time. The actual call is re-written to transfer control to the appropriate clone.

Chapter 5

Experiments and Results

5.1 Benchmark Suite of Secondary Threads

The theory behind the proposed techniques has been evaluated against the three possible deployment scenarios of ASTI as shown in *Figure 5.1*.

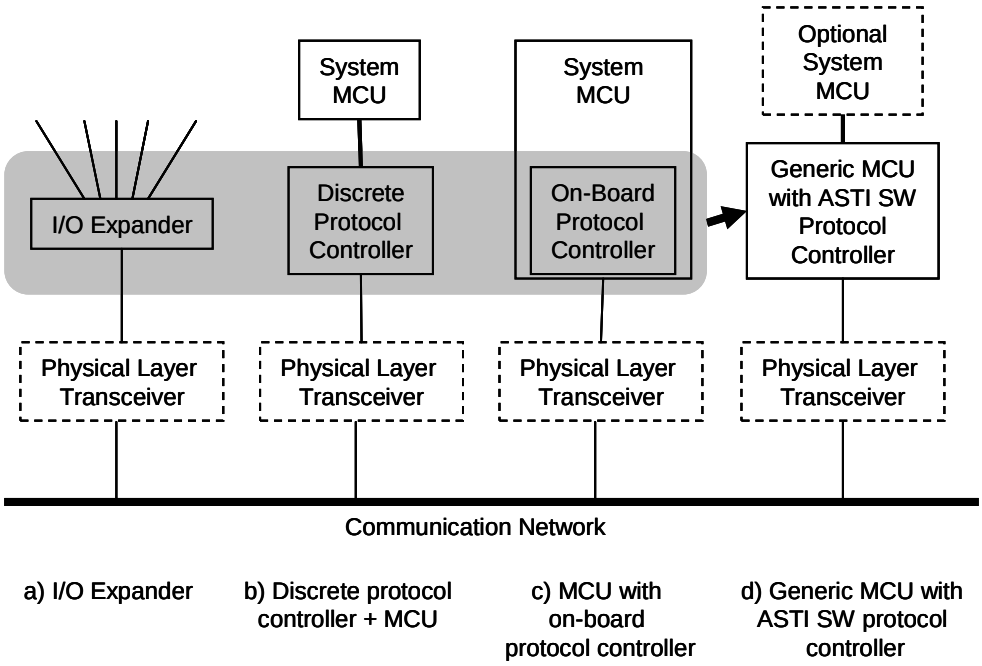


Fig.5.1 *Three types of hardware protocol controllers, and their replacement with a generic MCU.*

Figure 5.1a shows the simple "I/O expander" (or "Serially Linked I/O") device, which connects discrete analog and digital inputs and outputs to a network [Phil96, Micr01, Micl02, Phil02]. This device can act upon messages commanding certain output values, and can reply to requests for the state of the inputs. It can also automatically generate a message when specific input conditions occur. *Figure 5.1b* shows a discrete protocol controller, which acts as a bridge between the communication bus and the system's mi-

crocontroller, using either a serial (e.g. UART) or parallel link. *Figure 5.1c* shows a microcontroller with an on-board protocol controller; this is the most powerful option. *Figure 5.1d* shows the architecture of the system based on ASTI. The ASTI techniques are most suitable for replacing the I/O expander and stand-alone protocol controller. This is because integration complexity rises with application code size and the secondary threads have reduced real-time performance. However, a simple application could use ASTI to replace a microcontroller containing an integrated protocol controller with a generic (less expensive) microcontroller.

Corresponding to the three scenarios a benchmark suite of secondary threads was developed for evaluating the techniques presented in earlier sections. These benchmark threads are -

- A Smart node application – Software implementation of a subset of the functionalities found in [Phil96, Micr01, Micl02, Phil02], including individually configurable I/O ports, PWM channels, ADC channels and fan control. This corresponds to scenario *5.1a*.
- A host interface thread – Software thread receiving messages from the host microcontroller for transmission on the bus and transmitting received messages from the physical bus to the host microcontroller. Uses the serial UART for communicating with the host microcontroller. This secondary thread corresponds to scenario *5.1b*.
- A PID controller application – Simple closed loop proportional integral derivative control system. The feedback input is passed through an FIR filter for noise cancellation and then linearized before entering the PID control computation. This corresponds to scenario *5.1c*.

These secondary threads were implemented so as to be as generic as possible with respect to the communication protocol of the primary thread, which could be J1850, CAN or 1553.

5.2 Methodology of Experiments

Both phases one and two were applied to each of the secondary threads. In phase one, the inlining algorithm was executed for different values of the code expansion limit. The code expansion limit was specified as some percentage of the original. The limit was varied from 100 to 150 percent in steps of 5 percent each run. During each run of the experiment, a naïve and straightforward approach was used to calculate the code expansion due to an inlining operation - i.e. code expansion due to inlining a call site is assumed to be equal to the size of the procedure. In reality, this will be much less due to the elimination of the procedure's epilogue and prologue. The inlining decision was implemented and the actual code size, as a result of the inlining steps, was also measured.

In phase two, each set of source files, with some subset of procedures inlined according to phase one, was run through the phase two algorithm. Both the cocall duration and the cloning ratio were varied for each source file. The cocall duration was varied from as low as 50 cycles up to 400 cycles. This corresponds to realistic values of actual protocols as observed in [Kum02]. The cloning ratio for each source file was varied from 0.1 (the minimum) up to the maximum possible ratio of 1 and the resulting static padding cycles saved and code expansion due to each procedure and the application as a whole were profiled.

While reducing the amount of padding cycles added statically is beneficial, it is not really a clear indicator of the resulting speedup of the thread compared against the dynamic execution speed of the thread without cloning. Fully measuring the dynamic speed requires profiling each deterministic segment of the secondary thread. While it is within possible limits, such an analysis was not performed for this research. Instead reasonable approximations of the dynamic speedup have been evaluated. For each application, the overall speedup due to cloning as compared to not cloning has been measured manually for deterministic segments of execution in the application.

5.3 Results for Smart Node Application

Table 5.1 shows some of the features of the code that is handed down for constraint relaxation to phase one. The results from running phase 1 on the smart node application are given in *Table 5.2*.

Table 5.1 *Smart node - Source code information*

Original Number of Procedures	24
Original Code Size	5216 bytes
Original Total Static Call Sites	49

Table 5.2 *Smart node - Phase one results*

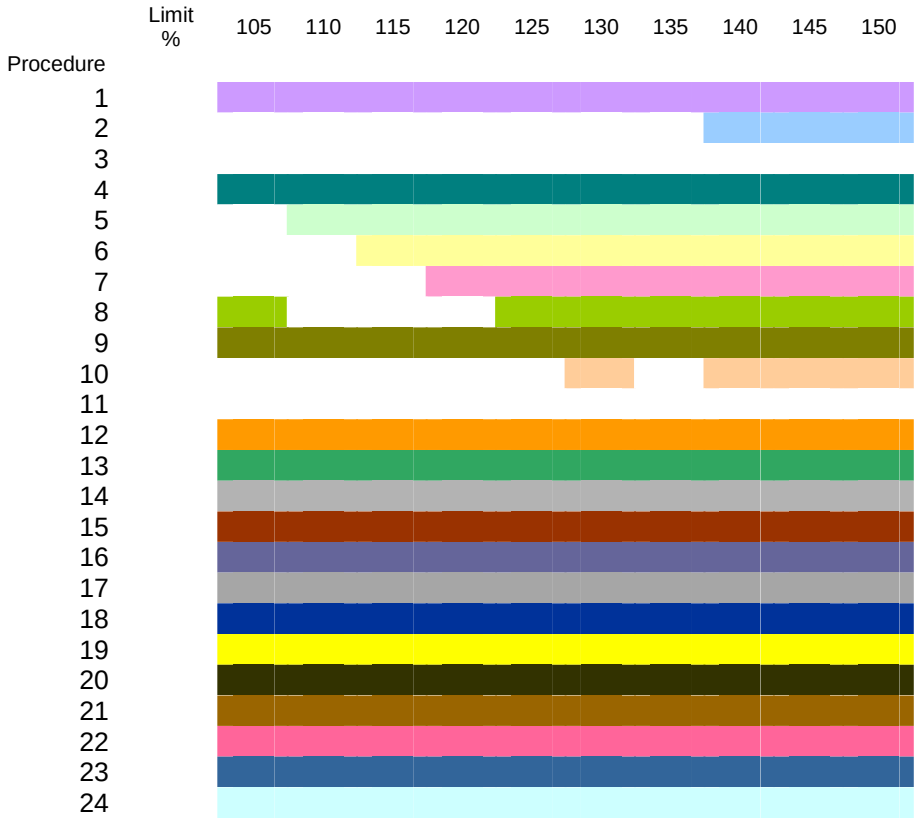
Code Explosion Limit (%)	Absolute Code Size Limit (bytes)	Expected Inlined Code Size (bytes)	Actual Code Size after inlining (bytes)	Remaining Procedures	Static call sites after inlining
105	5476	5386	4596	7	30
110	5737	5732	4516	7	26
115	5998	5956	4464	6	23
120	6259	6200	4552	5	21
125	6520	6294	4566	4	19
130	6780	6532	4580	3	17
135	7041	6890	4660	2	14
140	7302	7128	4660	2	14
145	6406	7128	4660	2	14
150	6627	7128	4660	2	14

Table 5.2 shows the following interesting variations. As soon as inlining is started, the number of procedures drops down to 7 from the original 24. This implies that most of the procedures in the baseline code can be inlined causing only minimal or no code expansion. Intuitively, these procedures service a single call site (*singletons*) or are very small in size and have very few incoming calls. In spite of a significant reduction in the number of procedures with the start of inlining, the number of static call sites still stays high. This is again because of a few large procedures which have many incoming calls into them. They tend to be utility procedures that get invoked from many places in the code. The other interesting thing to note is that actual resultant code size increases very slowly when compared to the rise in the expected code size with an increase in code ex-

pansion limit. This is because, the reduction in the procedure's prologue and epilogue cannot be predicted until the inlining operation actually takes place. In fact, the final code size tends to stay below the original size of the code as when it had procedures.

Table 5.3 indicates the inlining decisions of each procedure along the variation of the code expansion limit. As can be seen, a majority of the procedure stay inlined. These are procedures that have only one incoming call and therefore they will be chosen to be inlined no matter what the code expansion limit is. Some procedures (11 and 3) are always chosen to be not inlined, even when the code expansion limit is raised to 150% of the original. These are the procedures that have many incoming calls (7 calls each) to them. Combined with the significant code size of each of the procedure, inlining these, pushes the code expansion over the limit. These are exactly the procedures that need to be handled through cloning in phase two.

Table 5.3 *Smart node – Phase one inlining decisions*



The resulting code from phase one is sent through the phase two implementation. The amount of cloning is controlled by the clone ratio, as described earlier. A clone ratio of 0.5 is a point where there is a reasonable trade-off between code expansion and static padding cycles. This tradeoff is more clearly illustrated in *Figure 5.2*. It shows the effect of varying the clone ratio for many different cocall durations. As can be seen from the figure, a clone ratio of 0 or 0.1 represents no cloning or something close to that. As soon as we start allowing more clones, the amount of padding cycles added drops drastically.

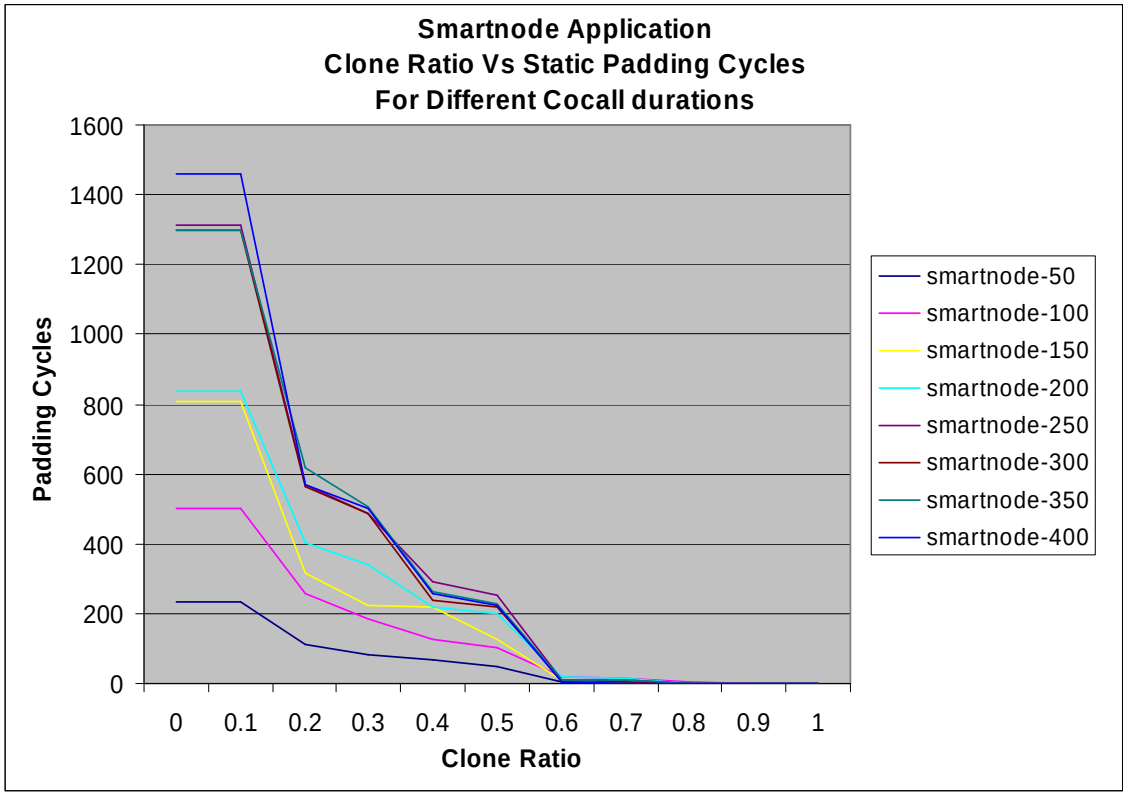


Fig.5.2 *Smart node - Tradeoff between static padding cycles and clone ratio*

The steepness of the drop reduces for lower cocall durations, which is logical because, with lower cocall durations, the amount by which variance of different Δc's from one another is greatly reduced at lower cocall durations. With further increase in cloning the savings keep improving. Static padding cycles are almost brought down to a negligible level by the time the cloning ratio reaches 0.5. An interesting observation is that the

cloning ratio needs to be raised no further than 0.65 to reach maximal savings. The same pattern follows for different cocall durations, with each curve having similar gradations.

The results for varying the cocall duration while keeping the cloning ratio at 0.5 are shown in *Figure 5.3*. These results are compared against not cloning the secondary thread. The amount of static padding cycles added increases rapidly with increase in the cocall duration. This is because there is a greater interval within which Δc's can be separated from each other. Therefore, padding all Δc's to the worst case, intuitively, becomes more and more expensive with larger cocall intervals.

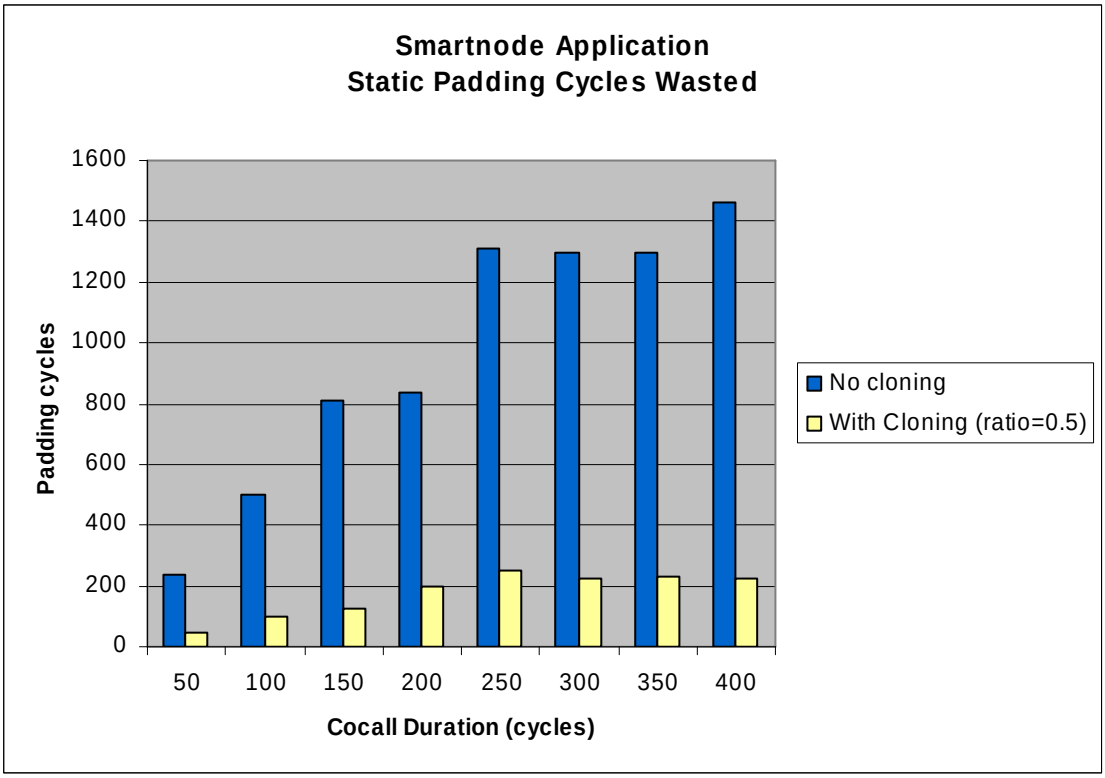


Fig.5.3 *Smart node – Cocall duration Vs static padding cycles*

Consequently, the amount of savings got by cloning the secondary thread, increases significantly with increase in the cocall interval. It is also interesting to note that the wasted cycles are brought down to comparatively good levels by cloning at a ratio of 0.5. The reason that the savings don't advance much beyond 200 cycles for higher cocall durations is because of the inability of the cloning algorithm to optimize a particular procedure which always runs into bad Δc distributions. That this is not the general trend of the

cloning algorithm can be seen by the same illustration against a single representative procedure *handle_error* in the smart node secondary thread, in *Figure 5.4*. As can be seen, whatever be the cocall duration, the amount of static padding cycles is always reduced to negligible levels for the procedure. This is the general trend for most of the procedures in the secondary thread. This again reaffirms the benefits of cloning.

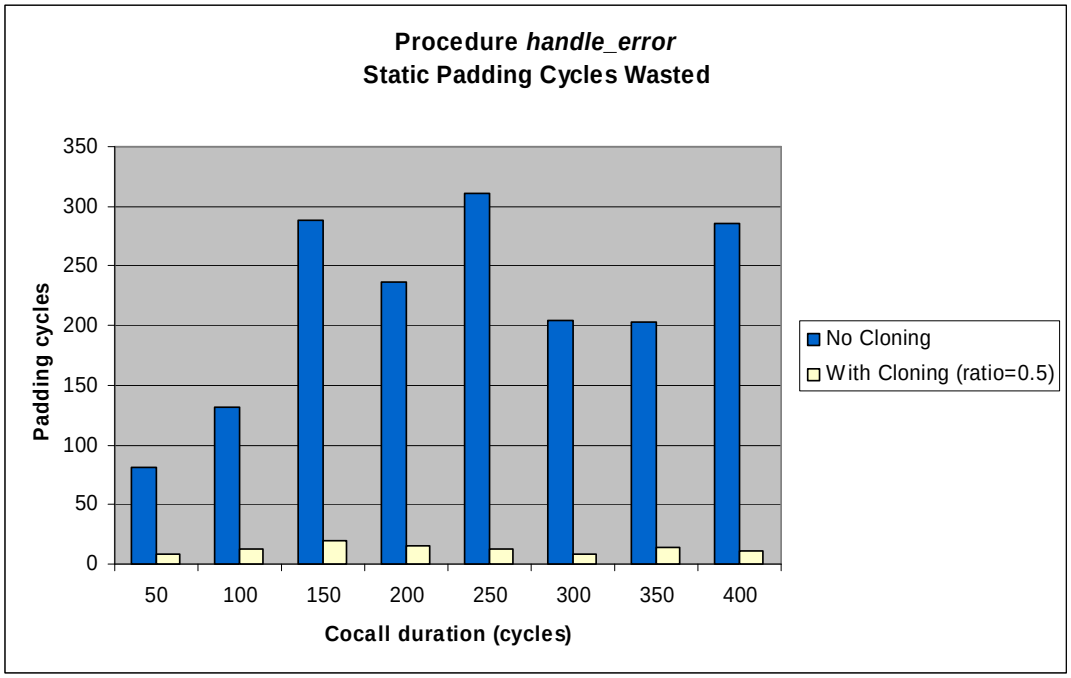


Fig.5.4 *Representative procedure from Smart node – cocall duration Vs static padding cycles*

5.4 Results for the PID Controller Application

Table 5.5 shows some of the features of the code that is handed down for constraint relaxation to phase one. The results from running phase one on the PID controller application are given in *Table 5.6*.

Table 5.5 *PID Controller – Source code information*

Original Number of Procedures	8
Original Code Size	1618 bytes
Original Total Static Call Sites	19

Table 5.6 *PID Controller – Phase one results*

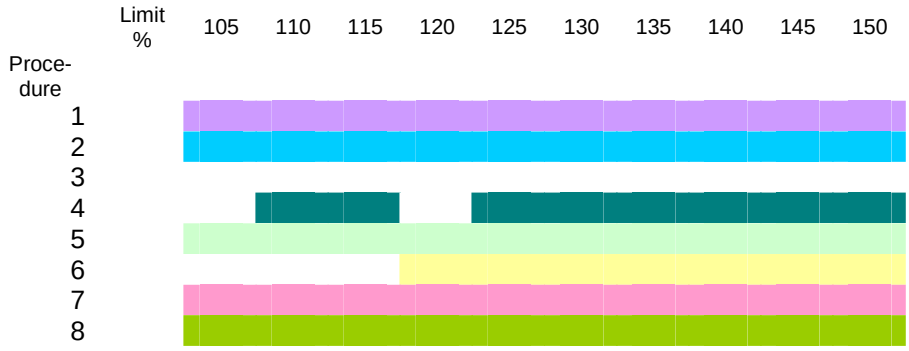
Code Explosion Limit (%)	Absolute Code Size Limit (bytes)	Expected Inlined Code Size (bytes)	Actual Code Size after inlining (bytes)	Remaining Procedures	Static call sites after inlining
105	1698	1618	1194	3	14
110	1779	1734	1142	2	11
115	1860	1734	1142	2	11
120	1941	1866	1162	2	9
125	2022	1982	1106	1	6
130	2103	1982	1106	1	6
135	2184	1982	1106	1	6
140	2265	1982	1106	1	6
145	2346	1982	1106	1	6
150	2427	1982	1106	1	6

Table 5.6 shows the following interesting variations. The PID controller application uses the 8.8 fixed-point representation to speed up arithmetic. The fixed point arithmetic routines such *fpmul, fpadd* and *fpsub* are often invoked from various places throughout the PID controller code and hence are the routines which have the most incoming calls. Of these three, the largest and most often invoked procedure in the application is the fixed-point multiply routine. As soon as inlining is started, the number of procedures drops down to 3 from the original 8. This is behavior is consistent with the fact that the rest of the routines *singletons* and hence are better off inlined. In spite of a significant reduction in the number of procedures with the start of inlining, the number of static call sites still stays high. This is again because of the fixed point routines being large procedures which have many incoming calls into them. Just as in the case of the smart node application, the resultant code size actually decreases with increase in the number of inlining operations. This is because of savings due to the elimination of the procedure epilogue and prologue.

Table 5.7 indicates the inlining decisions of each procedure along the variation of the code expansion limit. As can be seen a majority of the procedure stay inlined. These are

procedures that have only one incoming call and therefore they will be chosen to be inlined no matter what the code expansion limit is. The *fpmul* procedure (3) is always chosen to be not inlined, even when the code expansion limit is raised to 150% of the original. This is because it is a large procedure and has 6 incoming calls. This is an ideal procedure to be handled through cloning in phase two.

Table 5.7 *PID Controller – Phase one inlining decisions*



The resulting code from phase one is sent through the phase two implementation. *Figure 5.5* shows the effect of varying the clone ratio for many different cocall durations. As can be seen from the figure, a clone ratio of 0 or 0.1 represents no cloning or something close to that. As soon as we start allowing more clones, the amount of padding cycles added drops drastically. With further increase in cloning, the savings keep improving. Static padding cycles are almost brought down to a negligible level by the time the cloning ratio reaches 0.4. An interesting observation is that the cloning ratio needs to be raised no further than 0.6 to reach maximal savings. The same pattern follows for different cocall durations.

The results for varying the cocall duration while keeping the cloning ratio at 0.5 are shown in *Figure 5.6*. These results are compared against not cloning the secondary thread. The results show similar savings as in the smart node application. The amount of savings got by cloning the secondary thread, increases significantly with an increase in the cocall interval. No matter what the cocall duration, cloning at ratio 0.5 reduces the amount of static padding to a very low value. The same trend is seen in the savings for a single representative procedure *fpmul* in the secondary thread as shown in *Figure 5.7*.

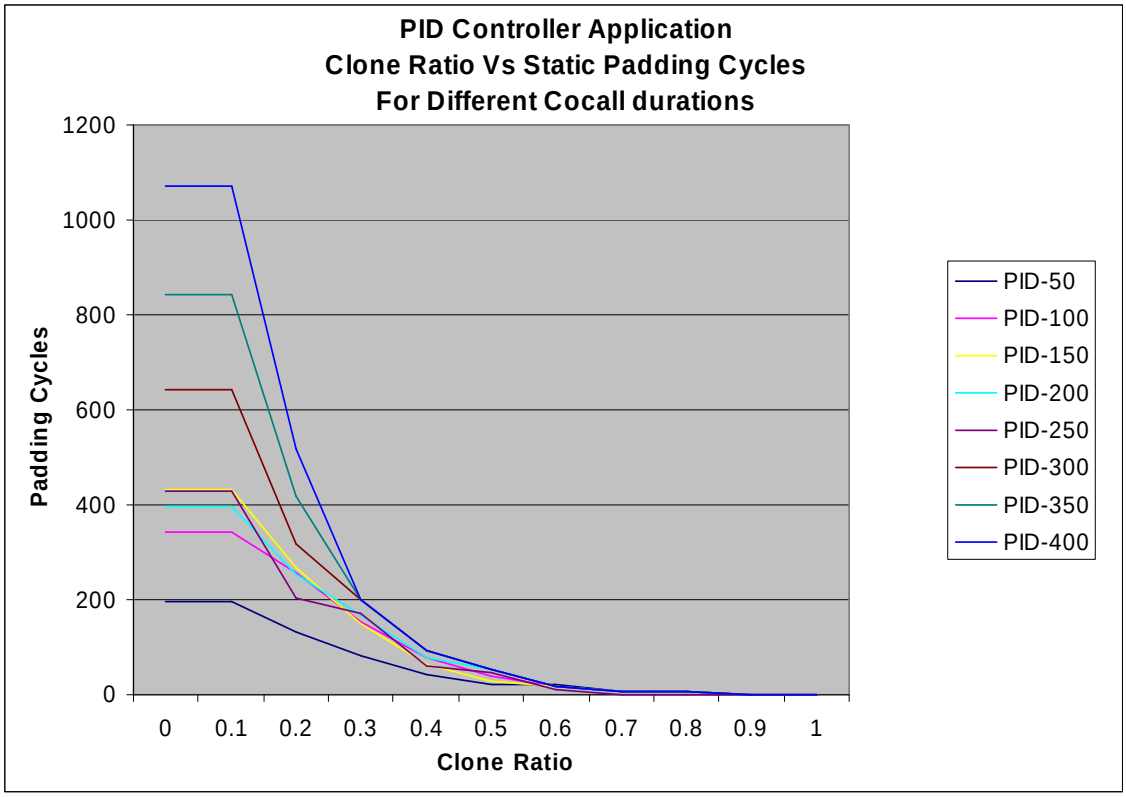


Fig.5.5 *PID Controller - Tradeoff between static padding cycles and clone ratio*

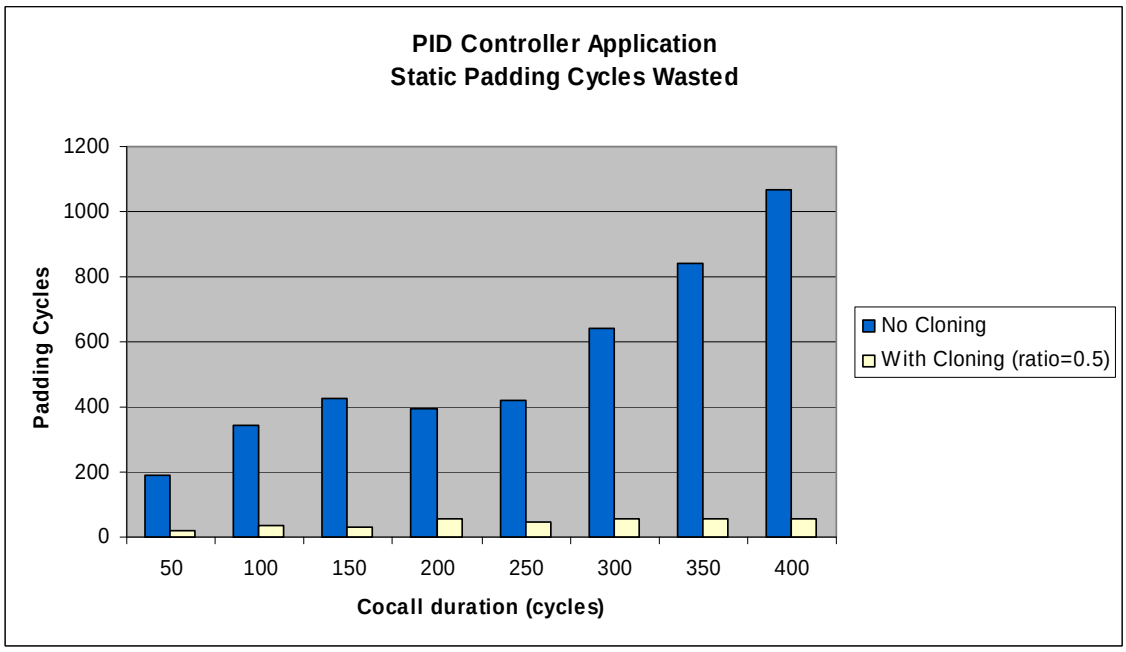


Fig.5.6 *PID controller – Cocall duration Vs static padding cycles*

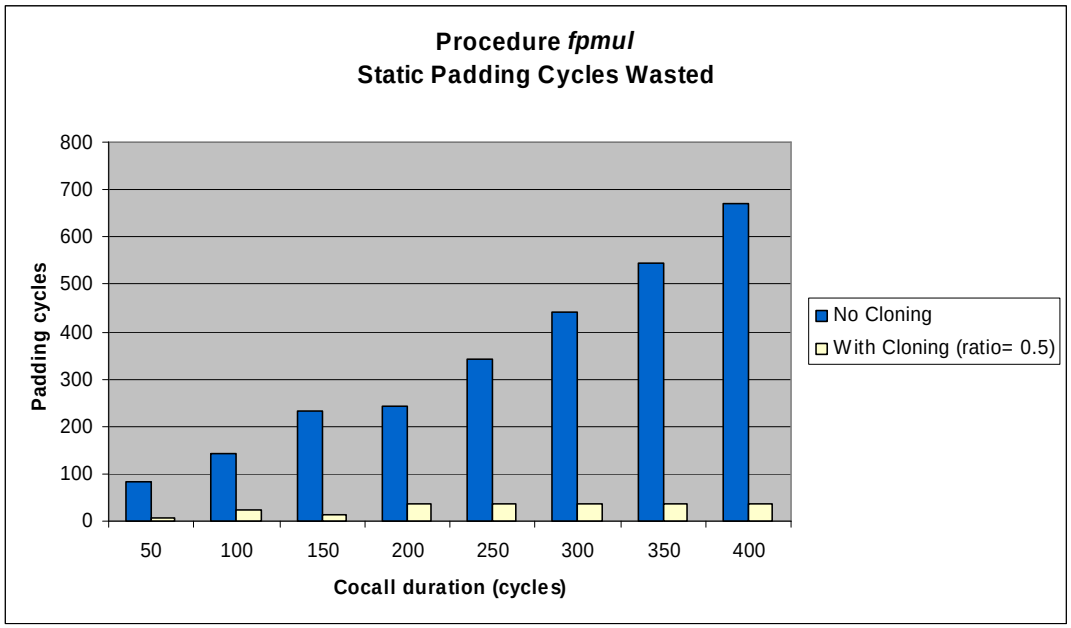


Fig.5.7 *Representative procedure from PID controller – cocall duration Vs static padding cycles*

5.5 Results for the Host Interface Thread

The host interface thread code is much simpler in terms of structure and hence simple, straightforward results are seen in both phase one and two. *Table 5.8* shows some of the features of the host interface thread that is handed down for constraint relaxation to phase one. The results from running phase 1 on the host interface thread are given in *Table 5.9*.

Table 5.8 *Host Interface Thread – Source code information*

Original Number of Procedures	11
Original Code Size	1472 bytes
Original Total Static Call Sites	18

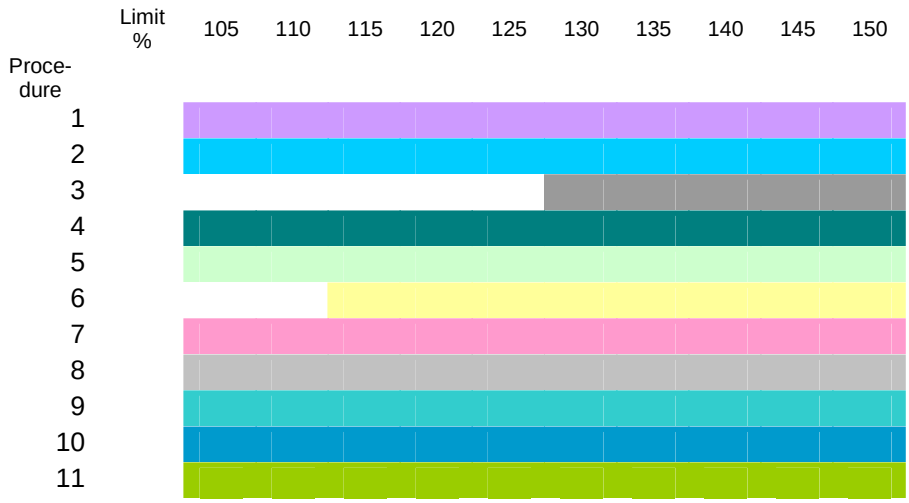
Table 5.9 *Host Interface Thread – Phase one results*

Code Explosion Limit (%)	Absolute Code Size Limit (bytes)	Expected Inlined Code Size (bytes)	Actual Code Size after inlining (bytes)	Remaining Procedures	Static call sites after inlining
105	1545	1472	1130	11	9
110	1619	1472	1130	2	9
115	1692	1688	1240	1	4
120	1766	1688	1240	1	4
125	1840	1688	1240	1	4
130	1913	1982	1106	0	0

The most often used routines are the routines used to get data from the UART receive buffer and to send data to the interface. These routines are not very large routines and hence they get inlined pretty quickly. By tuning the code expansion limit to only 130%, we find that all the procedures get inlined. Similar to the other phase one results, the actual code size after inlining is much better than the naïve estimate.

Table 5.10 indicates the inlining decisions of each procedure along the variation of the code expansion limit. As mentioned above, the host interface thread is pretty simple in terms of code structure. There are just two procedures with multiple incoming calls and hence most of the other procedures get inlined immediately.

Table 5.10 *Host Interface Thread – Phase one inlining decisions*



The resulting code from phase one is sent through the phase two implementation. *Figure 5.8* shows the effect of varying the clone ratio for many different cocall durations. As can be seen from the figure, a clone ratio of 0 or 0.1 represents no cloning or something close to that. As soon as we start allowing more clones, the amount of padding cycles added drops drastically. With further increase in cloning the savings keep improving. Static padding cycles are almost brought down to a negligible level by the time the cloning ratio reaches 0.4. An interesting observation is that the cloning ratio needs to be raised no further than 0.6 to reach maximal savings. The same pattern follows for different cocall durations.

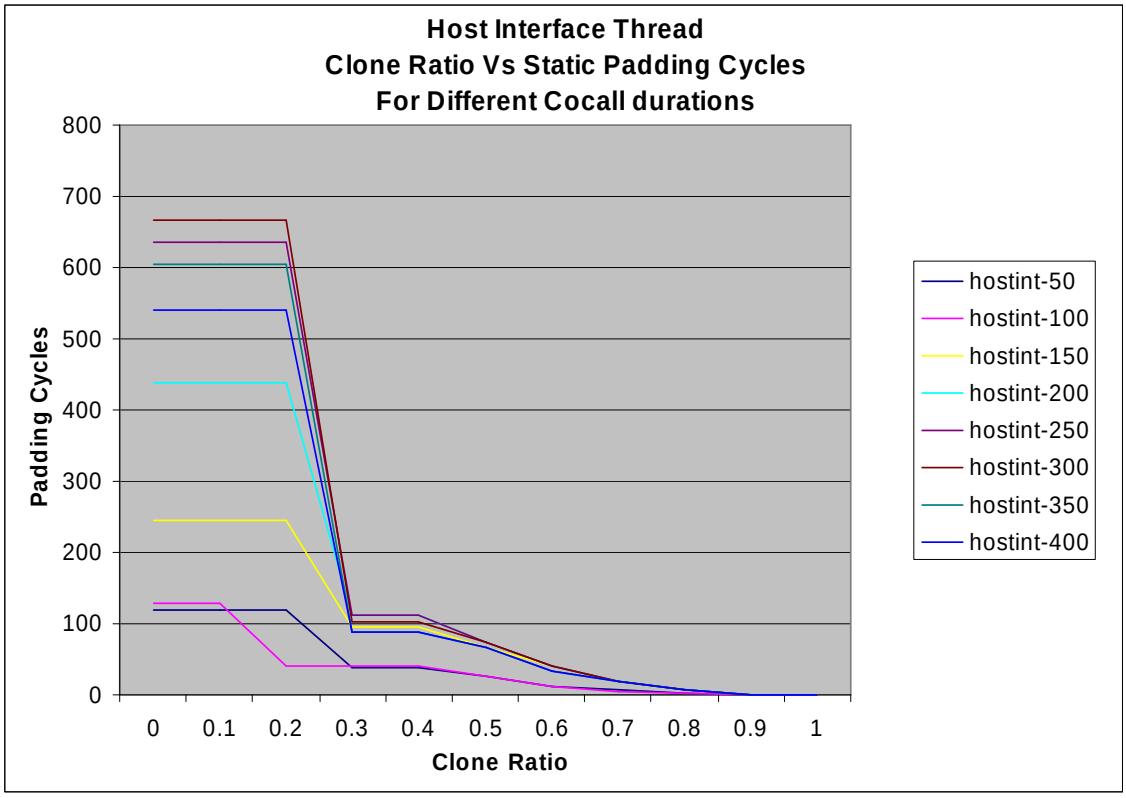


Fig.5.8 *Host interface thread - Tradeoff between static padding cycles and clone ratio*

The results for varying the cocall duration while keeping the cloning ratio at 0.5 are shown in *Figure 5.9*. These results are compared against not cloning the secondary thread. The results show similar savings as in the other two applications. The amount of

savings got by cloning the secondary thread, increases significantly with increase in the cocall interval.

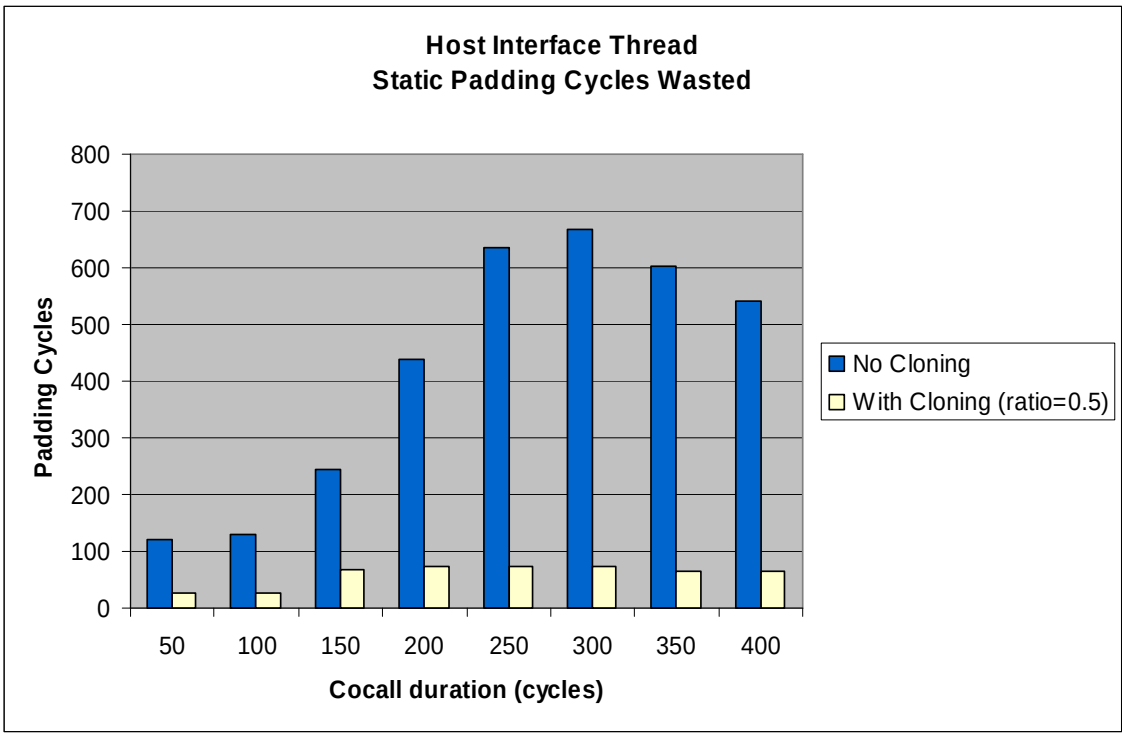


Fig.5.9 *Host interface thread – Cocall duration Vs static padding cycles*

5.6 Code Expansion Due to Cloning

While cloning provides major benefits in saving processor cycles as seen in the previous section, excessive cloning is akin to inlining all calls to a procedure and has the same code expansion effect. *Figure 5.10* shows the effect of varying the clone ratio on the resultant code expansion due to clones for the three benchmark secondary threads. Code growth is approximately linear with the increase in the cloning ratio. The expansion for the PID controller application and the host interface thread are much more sedate when compared to the smart node application. This can be attributed to the higher complexity and baseline code structure of the smart node application.

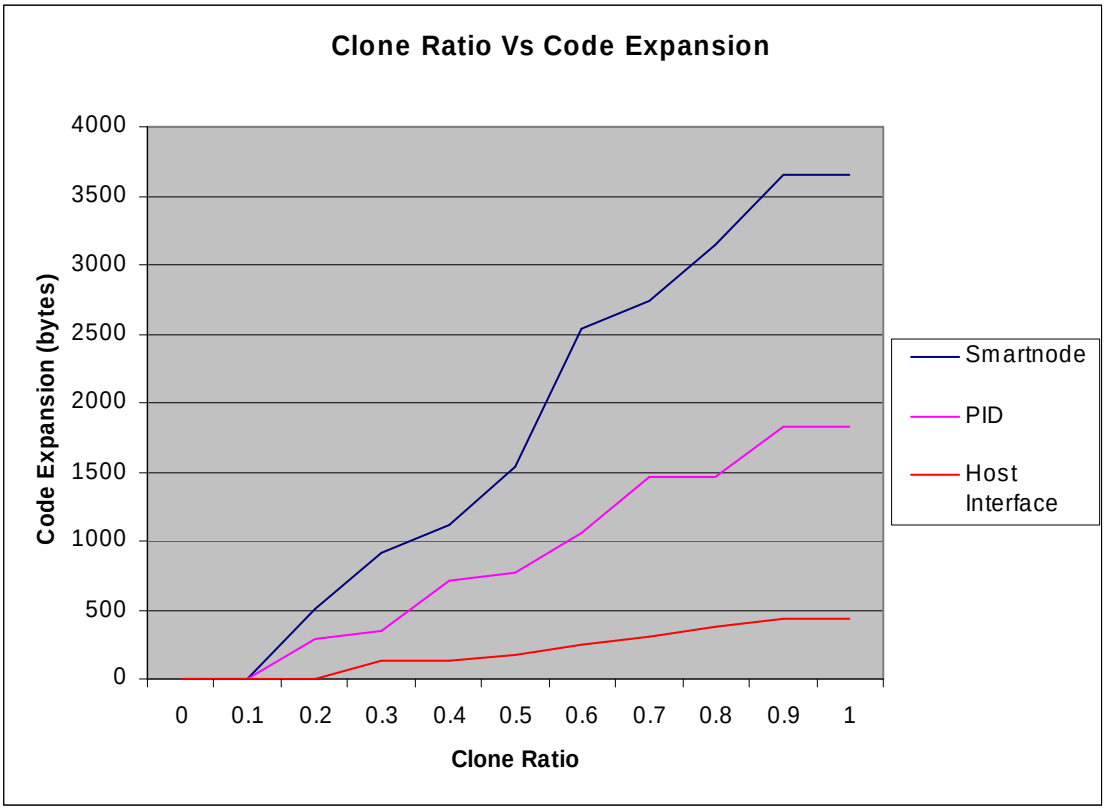


Fig.5.10 *Increase in code expansion with changes in clone ratio*

The expansion as a percentage of the original code size, at high cloning ratios (> 0.8) is generally at unacceptable levels except for the host interface thread. But fortunately as illustrated in the previous results, we do not need to tune the cloning ratio to such high levels. Significant savings were illustrated even at a cloning ratio of 0.5. In general, based on the configuration of the target micro-controller and the deployment scenario, the trade-offs between code size and saved padding cycles can be evaluated and a suitable cloning level can be chosen.

5.7 Speedup over non-cloned versions due to cloning

Static padding cycles do not accurately reflect the increase in the processor efficiency at run-time. To accurately measure speedup due to cloning as compared against the non-cloned versions of the threads, the dynamic behavior of the code needs to be known. Dynamic profiling is needed to determine the frequency of invocation of procedures and

to find out places where most of the execution is spent in. While profiling techniques to estimate these details to a highly accurate degree exist, they have not been applied in this thesis. Instead we focus on an analysis of deterministic code fragments in the secondary threads. Each of the secondary threads has an outermost infinite while loop. Taking each iteration of the body of the while loop to be a single execution of the secondary thread, we estimate cycles spent in executing the cloned version and the non-cloned version of the loop body. The smart node application is structured as a state machine, which based on the commands that it receives, performs one of many different operations. In this case, unless we have an idea about the dynamic nature of the commands received, the exact operations that will be performed at run-time cannot be estimated. Therefore one representative deterministic segment from one particular state of the finite state machine has been chosen for analysis and the speedup due to cloning in that particular segment alone has been estimated. Though this is not an accurate estimation of speedup, it serves as a roughly accurate indicator of the benefits of cloning. It must be noted that obtained values are speedups over the version of the secondary that has procedures but does not choose to clone them to optimize padding cycles. They are not indicators of speedups over the existing secondary threads which do not have procedures at all.

Figure 5.11 shows the measured speedup of the smart node application and the PID controller application. As can be seen from the figure, the speedup obtained is significant when the cocall duration is higher. The effects of padding cycles get amplified at higher cocall durations because of the presence of loops. Calls that are under loops will cause potential padding cycles to be inserted under the loop and hence cycles will be wasted each loop iteration. Therefore the static padding cycles count gets multiplied by the iteration count of the loop. The generally low speedup seen in the PID controller application is because of two reasons – one is that only two calls appear within loop bodies and the second reason is that the total execution cycles taken by the PID controller loop is very high, compared to the number cycles saved. The high cycle count is mainly due to the execution complexity of the fixed point multiply routine which gets invoked often.

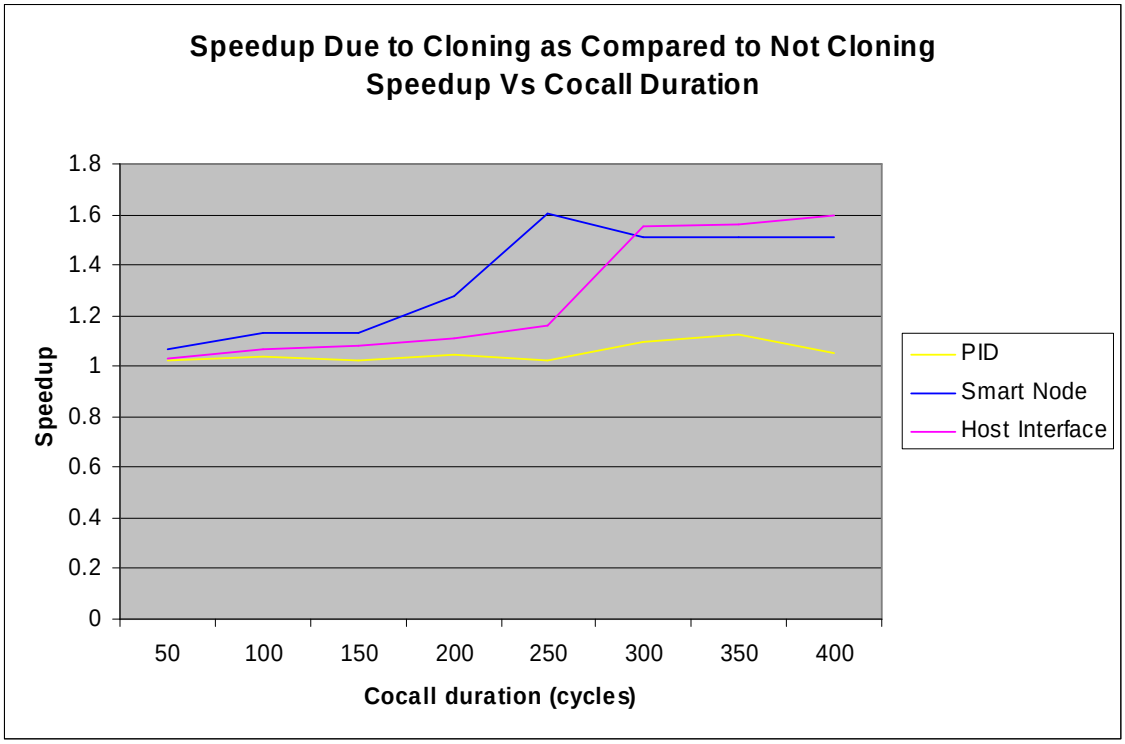


Fig.5.11 *Speedup due to cloning*

The effect of loops on padding cycles gets exacerbated at large cocall durations. This results in speedups of the order 50-60%. This effect is seen in both the host interface thread and the smart node application. It must be noted that since these speedups have been evaluated only on deterministic segments in the body of entire secondary thread, they represent only *fractional* speedup. The overall speedup can only be measured by dynamic profiling as mentioned earlier and will be generally lower than the fractional speedup, when all the execution segments of the thread are considered.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Relaxing control flow constraints in existing ASTI techniques is very important as it otherwise limits the usefulness and applicability of the methods. This thesis has presented a step-by-step solution to relax control flow constraints in secondary threads, by applying the traditional compiler optimization techniques – procedure inlining and procedure cloning, in a refreshingly different context. The techniques presented provide an algorithmic basis for trading off code size and processor efficiency. Clustering procedure calls invoked in similar contexts, together, results in significant savings in the amount of wasted padding code that is required. These savings will reflect in the dynamic behavior of the secondary thread, thus allowing more secondary work to be performed as compared to the approach which does not use cloning. This is highly desirable as it will allow system designers to not raise clock speeds to improve the real-time performance of the secondary threads. A scientific control flow relaxation process also allows code writers to be abstracted away from details of the later STI phase. Code writers need not write flat code and are now free to use procedures. Such code is easier to maintain, debug and develop. By demonstrating the techniques against the three ASTI deployment scenarios – the smart node application, the host interface thread and the PID controller, we contend the usefulness of the techniques presented in this thesis.

6.2 Future Work

The methods presented can be generalized by incorporating techniques to remove some assumptions made. Recursive procedures are currently handled by neither the inlining nor the cloning phase. The techniques can be modified to inline recursive procedures to

some tolerable levels. The naïve estimation of the effect of an inlining step can be modified to take into account the elimination of the procedure's prologue and epilogue. This will allow us to achieve a desired amount of inlining without having to raise the code size limit to higher levels. The optimal inlining algorithm presented takes an all or none approach towards inlining a procedure – i.e. either all calls are inlined or none are. This can be relaxed to allow the algorithm to distinguish and choose between individual incoming calls of a procedure based on some other guiding criteria, like dynamic call frequency information etc. This will lead to more optimal code as a result of traditional inlining benefits. As described earlier an optional PRE phase between phases one and two of the constraint relaxation algorithm could provide significant savings. Further research is needed to actually gauge its worth. Padding of procedures to last the duration of a cocall represents a step in the backward direction and finding techniques to work around having to do such padding will produce code that has improved real-time performance. Such a solution requires an approach which looks at call sites one by one in the control flow order, rather than holistically as done in this thesis. The current research can be extended to handle multiple primary threads with just a single copy of the procedure body. This will greatly reduce the code expansion produced by multiple copies of the secondary threads.

References

[LM99] Rainer Leupers, Peter Marwedel, "Function Inlining Under Code Size Constraints for Embedded Processors". *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, p.253-256, November 07-11, 1999, San Jose, California, United States

[KSD03] Nagendra J. Kumar, Siddhartha Shivshankar and Alexander G. Dean. "Asynchronous Software Threading for Efficient Software Implementations of Embedded Communication Protocol Controllers". Center for Embedded Systems Research, NCSU, 2003.

[Kum02] Nagendra J. Kumar, "STI concepts for bit-bang communication protocols". Masters Thesis, North Carolina State University, May 2003.

[Dea00] Alexander G. Dean, "Software Thread Integration for Hardware to Software Migration". Doctoral Dissertation, Carnegie-Mellon University.

[Dea03] Alexander G. Dean, Thrint Manual, Release 1.0, http://www.cesr.ncsu.edu/projects/sti/websti/home.html

[RG89] S. Richardson and M. Ganapathi, "Code Optimization Across Procedures". *IEEE Computer*, February 1989.

[CMCH92] Pohua P. Chang, Scott A. Mahlke, William Y. Chen and Wen-mei W. Hwu, "Profile-guided Automatic Inline Expansion for C Programs". *Software ---Practice and Experience*, 25:349-369, 1992.

[AFSS00] M. Arnold, S. Fink, V. Sarkar and P. Sweeney, "A Comparative Study of Dynamic and Static Heuristics for Inlining". *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, 2000.

[AC72] F.E. Allen and J. Cocke, "A Catalogue of Optimizing Transformations". in *Design and Optimization of Compilers*, pages 1-30, Prentice Hall, 1972.

[MR79] Etienne Morel and Claude Renvoise, "Global Optimization by the Suppression of partial redundancies", *Communications of the ACM*, 22(2):96-103, February, 1979.

[PS82] D.A. Patterson and C. H. Sequin, "A VLSI RISC". *IEEE Computer*, pp 8-21, September, 1982.

[KR98] Owen Kaser and C. R. Ramakrishnan, “Evaluating Inlining Techniques”. *Computer Languages*, 24:45-72, 1998

[CHK89] K.D. Cooper, M.W. Hall and K. Kennedy, “A Methodology for Procedure Cloning”. *Computer Languages,* pages 146-160, 1989.

[GJ79] Michael R. Garey and David S. Johnson, “Computers and Intractability: A Guide to the theory of NP-Completeness”. *W.H. Freeman and Company*, 1979.

[DCG95] J. Dean, C. Chambers and D. Grove, “Selective Specialization for Object-Oriented Languages”. In ACM SIGPLAN, Conference on Programming Languages Design and Implementation, pp 93-102, June 1995.

[SD03] Won So and Alexander G. Dean, "Procedure Cloning and Integration for Converting Parallelism from Coarse to Fine Grain". *7th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-7, in conjunction with HPCA 9)*, February 8, 2003, Anaheim, CA

[FOW87] Jean Ferrante, Karl J. Ottenstein and Joe D. Warren, “The Program Dependence Graph and Its Use in Optimization”. *ACM Transactions on Programming Languages*, 9(3), July 1987, 319-349.

[Phil96] Philips Semiconductors, “P82C150 CAN Serial Linked I/O Device (SLIO) with Digital and Analog Port Functions Data Sheet”. June 19, 1996

[Micr01] Microchip Technology, Inc. “MCP2502X/5X CAN I/O Expander Family Data Sheet”. 2001

[Micl02] Micrel, Inc. “MIC74 2-Wire Serial I/O Expander and Fan Controller”. August 2000

[Phil02] Philips Semiconductors, “PCF8574 Remote 8-bit I/O Expander for I^2C-bus”. November 22, 2002

Appendix 1

Source Code for the Benchmark Secondary Threads

1. Smart Node Application Code

```
/*
 * Smart Node Functionality - Benchmark Suite Secondary Thread
 * This is the original smart node code -
 *                       Non-flat, procedurized version.
 * Target Microcontroller   - Atmel Atmega103
 */


#include "smartnode.h"

// Global Data

extern SN_Msg recv_buf[BUF_MAX];
extern SN_Msg send_buf[BUF_MAX];
extern int recv_queue_head, recv_queue_tail;
extern bool recv_queue_contains_data;
extern int send_queue_head, send_queue_tail;
extern bool send_queue_contains_data;
extern bool send_queue_ovflow; // Flag to indicate that the send queue
            has overflowed.
                              // It is the primary threads job to re-
            set this flag appropriately
                              // And take error reporting action.

u16 DataIn, DataOut, DataDirection;  // I/O Port registers
u16 FanSpeed,PrevFanSpeed;           // Fan Speed register
u16 Event_Capture_Mask;              // Event Capture Register

// Misc vars
SN_Msg cur_send_msg;

// Definitions
//-----------------------------------------------------------------
// Main Control Loop
//-----------------------------------------------------------------
void main_control_loop()
{
    SN_Msg cur;

    init_smart_node();

    cur_send_msg.type = SN_TYPE_ACK;
    cur_send_msg.func = SN_FUNC_ACK_STARTUP;
    send_acknowledgement(&cur_send_msg,0);

    // Need to do the following in an infinite loop
```

```
    while(1) { // Take out to avoid unstructured code

        // Check the recv queue for any messages
        if( dequeue(RECV_QUEUE,&cur) != FALSE)
            handle_message(&cur);
        // Check each module. Enqueue necessary messages
        // Right now only the Event Capture module has to be watched.
        check_for_events();

    } // End infinite while
}

//-------------------------------------------------------
// Power Up Initialization
//-------------------------------------------------------
extern inline void init_smart_node() // Want this inlined always
{
    init_io_ports();
    init_pwm();
    init_adc();
    init_fan();
    init_power_mode();
    Event_Capture_Mask = 0;

    recv_queue_head = recv_queue_tail = 0;
    recv_queue_contains_data = FALSE;

    send_queue_head = send_queue_tail = 0;
    send_queue_contains_data = FALSE;

    send_queue_ovflow = FALSE;
}


//-----------------------------------------------------------------
// Parse Messages and Take actions
//-----------------------------------------------------------------
void handle_message(SN_Msg *cur)
{

    u08 *byte_param_p;
    u16 *word_param_p;
    u32 *dword_param_p;

    u16 word_ret;

    bool flag1, flag2;


    if( cur->type == SN_TYPE_SN) {
        __asm__("nop");
    }
    else if(cur->type == SN_TYPE_IOPORT) {
        handle_ioport(cur); // Need to delegate work to another subrou-
            tine, to avoid unstructured code.
    }
    else if(cur->type == SN_TYPE_PWM) {
```

```
        handle_pwm(cur);
    }
    else if(cur->type == SN_TYPE_ADC) {
        handle_adc(cur);
    }
    else if(cur->type == SN_TYPE_FAN) {
        handle_fan(cur);
    }
    else if( cur->type == SN_TYPE_POWER ) {
        handle_power(cur);
    }
    else if( cur->type == SN_TYPE_EVENT) {
        handle_event(cur);
    }
    else {
        // Bad request. Error response
        handle_error(cur,SN_FUNC_ACK_BAD_HDR);
    }
}

void handle_adc(SN_Msg *cur)
{
    u08 *byte_param_p;
    u16 *word_param_p;
    u32 *dword_param_p;

    u16 byte_ret;
    u16 word_ret;
    u16 dword_ret;


    bool flag_ok =  FALSE;
    u08 error_code = SN_FUNC_ACK_BAD_HDR;

    if( cur->func == SN_FUNC_ADC_INIT ) {
        init_adc();
        flag_ok = TRUE;
    }
    else if( cur->func == SN_FUNC_ADC_CHANNEL) {
        byte_param_p = (u08*)& (cur->params[0]);
        if( *byte_param_p > 15 ) { // Check for  param validity
            error_code = SN_FUNC_ACK_BAD_PARAM;
        }
        else {
            adc_setchannel(*byte_param_p);
            flag_ok = TRUE;
        }
    }
    else if( cur->func == SN_FUNC_ADC_ENABLE ) {
        adc_enable();
        flag_ok = TRUE;
    }
    else if( cur->func == SN_FUNC_ADC_DISABLE ) {
        adc_disable();
        flag_ok = TRUE;
    }
    else if( cur->func == SN_FUNC_ADC_PRESCALER ) {
```

```
        byte_param_p = (u08*)& (cur->params[0]);
        if( *byte_param_p > 7) { // Check for  param validity
            error_code = SN_FUNC_ACK_BAD_PARAM;
        }
        else {
            adc_setprescaler(*byte_param_p);
            flag_ok = TRUE;
        }
    }
    else if( cur->func == SN_FUNC_ADC_CONVERT ) {
        word_ret = adc_convert();
        flag_ok = TRUE;
    }

    if( flag_ok )
        send_acknowledgement(cur,word_ret);
    else {
        // Handle error condition here
        handle_error(cur,error_code);
    }
}

void handle_ioport(SN_Msg *cur)
{
    u08 *byte_param_p;
    u16 *word_param_p;
    u32 *dword_param_p;

    u16 word_ret;

    bool flag_ok = FALSE;
    u08 error_code = SN_FUNC_ACK_BAD_HDR;

    if( cur->func == SN_FUNC_IOPORT_INIT) {
        init_io_ports();
    }

    else if( cur->func == SN_FUNC_IOPORTS_CONFIG) {
        // Param: u16 - Individual bits indicating the direction of
            each IO Port
        word_param_p = (u16*)& (cur->params[0]);
        set_data_direction(*word_param_p);
    }

    else if( cur->func == SN_FUNC_IOPORT_CONFIG) {
        // Param: u08,u08 - Port, Direction
        byte_param_p = (u08*)& (cur->params[0]);
        if( *byte_param_p > 15 ) { // Also check for next param valid-
            ity
            error_code = SN_FUNC_ACK_BAD_PARAM;
        }
        else {
            configure_bit(*byte_param_p, *(byte_param_p+1));
            flag_ok = TRUE;
        }
    }
    else if( cur->func == SN_FUNC_IOPORT_READ_PORTS) {
```

```
        word_ret = read_data_register();
        flag_ok = TRUE;
    }
    else if( cur->func == SN_FUNC_IOPORT_READ_PORT) {
        byte_param_p = (u08*)cur->params;
        if( *byte_param_p > 15 ) {
            error_code = SN_FUNC_ACK_BAD_PARAM;
        }
        else {
            word_ret = read_bit_value(*byte_param_p);
            flag_ok = TRUE;
        }
    }
    else if( cur->func == SN_FUNC_IOPORT_SET_DATA) {
        word_param_p = (u16*)& (cur->params[0]);
        write_data_register(*word_param_p);
        flag_ok = TRUE;
    }
    else if( cur->func == SN_FUNC_IOPORT_SET_PORT) {
        byte_param_p = (u08*)& (cur->params[0]);
        if( *byte_param_p > 15 ) {
            error_code = SN_FUNC_ACK_BAD_PARAM;
        }
        else {
            write_bit_value(*byte_param_p,TRUE);
            flag_ok = TRUE;
        }
    }

    else if( cur->func == SN_FUNC_IOPORT_UNSET_PORT) {
        byte_param_p = (u08*)& (cur->params[0]);
        if( *byte_param_p > 15 ) {
            error_code = SN_FUNC_ACK_BAD_PARAM;
        }
        else {
            write_bit_value(*byte_param_p,FALSE);
            flag_ok = TRUE;
        }
    }

    if( flag_ok )
        send_acknowledgement(cur,word_ret);
    else {
        // Handle error condition here
        handle_error(cur,error_code);
    }
}

void handle_pwm(SN_Msg *cur)
{
    u08 *byte_param_p;
    u16 *word_param_p;
    u32 *dword_param_p;

    u16 word_ret;

    bool flag_ok = FALSE;
```

```
    u08 error_code = SN_FUNC_ACK_BAD_HDR;
    u08 flag1, flag2;

    if( cur->func == SN_FUNC_PWM_INIT) {
        init_pwm();
        flag_ok = TRUE;
    }
    else if( cur->func == SN_FUNC_PWM_START) {
        byte_param_p = (u08*)&(cur->params[0]);
        flag1 = (*byte_param_p != PWM_0);
        flag2 = (*byte_param_p != PWM_1);
        flag1 |= flag2; // Hacky. Need this to avoid generating un-
            structured code.
        if( flag1) { // Checking to see if the 'pwm' parameter is in
            order
            error_code = SN_FUNC_ACK_BAD_PARAM;
        }
        else {
            enable_pwm(*byte_param_p,               *(byte_param_p+1),
            *(byte_param_p+2));
            flag_ok = TRUE;
        }
    }
    else if( cur->func == SN_FUNC_PWM_STOP ) {
        byte_param_p = (u08*)&(cur->params[0]);
        flag1 = (*byte_param_p != PWM_0);
        flag2 = (*byte_param_p != PWM_1);
        flag1 |= flag2; // Hacky. Need this to avoid generating un-
            structured code.
        if( flag1) { // Checking to see if the 'pwm' parameter is in
            order
            error_code = SN_FUNC_ACK_BAD_PARAM;
        }
        else {
            disable_pwm(0x1);
            flag_ok = TRUE;
        }
    }

    if( flag_ok )
        send_acknowledgement(cur,word_ret);
    else {
        // Handle error condition here.
        handle_error(cur,SN_FUNC_ACK_BAD_HDR);
    }
}

void handle_fan(SN_Msg *cur)
{
    u08 *byte_param_p;
    u16 *word_param_p;
    u32 *dword_param_p;

    u16 byte_ret;
    u16 word_ret;
    u16 dword_ret;
```

```
    bool flag_ok = FALSE;
    u08 error_code = SN_FUNC_ACK_BAD_HDR;

    if( cur->func == SN_FUNC_FAN_INIT) {
        init_fan();
        flag_ok = TRUE;
    }
    else if( cur->func == SN_FUNC_FAN_SHUTDOWN) {
        shutdown_fan();
        flag_ok = TRUE;
    }
    else if( cur->func == SN_FUNC_FAN_SPEED) {
        byte_param_p = (u08*)& (cur->params[0]);
        if( *byte_param_p > 7 ) {
            error_code = SN_FUNC_ACK_BAD_PARAM;
        }
        else  {
            set_fan_speed(0x10);
            flag_ok = TRUE;
        }
    }

    if( flag_ok )
        send_acknowledgement(cur,word_ret);
    else {
        // Handle error condition here.
        handle_error(cur, error_code);
    }
}

void handle_power(SN_Msg *cur)
{
    u08 *byte_param_p;
    u16 *word_param_p;
    u32 *dword_param_p;

    u16 word_ret;

    bool flag_ok = FALSE;
    u08 error_code = SN_FUNC_ACK_BAD_HDR;

    if( cur->func == SN_FUNC_POWER_INIT) {
        init_power_mode();
        flag_ok = TRUE;
    }
    else if( cur->func == SN_FUNC_POWER_SLEEP) {
        byte_param_p = (u08*)& (cur->params[0]);
        if( *byte_param_p > 2 ) {
            error_code = SN_FUNC_ACK_BAD_PARAM;
        }
        else  {
            change_power_mode(*byte_param_p);
            flag_ok = TRUE;
        }
    }

    if( flag_ok )
```

```
        send_acknowledgement(cur,word_ret);
    else {
        // Handle error condition here.
        handle_error(cur,error_code);
    }


}

void handle_event(SN_Msg *cur)
{
    u08 *byte_param_p;
    u16 *word_param_p;
    u32 *dword_param_p;

    u16 word_ret;

    bool flag_ok = FALSE;

    if( cur->func == SN_FUNC_EVENT_INIT) {
        Event_Capture_Mask = 0; // Reset all event capture flags
    }
    else if( cur->func == SN_FUNC_EVENT_CONFIG ) {
        word_param_p  = (u16*)cur->params;
        Event_Capture_Mask = *word_param_p;
        flag_ok = TRUE;
    }
    if( flag_ok )
        send_acknowledgement(cur,word_ret);
    else {
        handle_error(cur,SN_FUNC_ACK_BAD_HDR);
    }
}

//-----------------------------------------------------------
// Primary Thread Interface
//-----------------------------------------------------------

//-----------------------------------------------------------
//  1. I/O Port Control
//  -   Totally offer 16 configurable I/O Ports.
//  -   Implemented using I/O Ports A,E.
//  -   Mapping: PORTA: P15-P8, PORTE: P7-P0
//-----------------------------------------------------------
extern inline void init_io_ports() // Want this inlined always
{
    outp(0, DDRE);  // Set as Input
    outp(0, DDRA);  // Set as Input
    DataIn = DataOut = DataDirection = 0;
}

void configure_bit( int io_port, int direction)
{
    int port, bit, ddr_port;

    get_port_and_bit(io_port,&port,&bit);
```

```
    if( port == PORTA )
        ddr_port = DDRA;
    else
        ddr_port = DDRE;

    if( direction == IN)
        cbi(ddr_port, bit);
    else
        sbi(ddr_port, bit);
}

extern inline int read_bit_value( int io_port) // Want this inlined al-
            ways
{
    int port, bit;
    u08 portval, val;
    get_port_and_bit(io_port,&port,&bit);

    portval = inp(port);
    return get_bit_value((u16)portval,bit);

}

void write_bit_value(int io_port, int val)
{
    int port, bit;
    get_port_and_bit(io_port,&port,&bit);

    if( val == 1 )
        sbi(port,bit);
    else
        cbi(port,bit);

    bit++; // dummy :(
    bit--; // Thrint complains sometimes

}

void get_port_and_bit(int io_port, int *port, int *bit)
{
    if( io_port <= 7 ) {
        *port = PORTE;
        *bit = io_port;
    }
    else {
        *port = PORTA;
        *bit = io_port - 8;
    }
}


extern inline void set_data_direction( u16 value) // Want this inlined
            always
{
    int i;
    int direction;
```

```
    DataDirection = value;

    for(i=0; i<16;i++) {
        direction = get_bit_value(value,i);
        configure_bit(i,direction);
    }

}

// Read and latch input port values into DataIn
// Return DataIn
u16 read_data_register()
{

    int i, val;
    u16 lower;

    DataIn = 0; // Purge old value.

    DataIn = inp(PINA);
    DataIn <<= 8;
    lower = inp(PINE);
    DataIn |= lower;

    return DataIn;
}

// Write value into DataOut
// Drive enabled output ports to corresponding values
void write_data_register( u16 value)
{
    int i;

    DataOut = value ;

    for(i=0;i<8;i++) {
        if( get_bit_value(DataDirection,i) == OUT ) { // An output en-
            abled port
            if( get_bit_value(DataOut,i) == TRUE )
                sbi(PORTE,i);
            else
                cbi(PORTE,i);
        }
    }

    for(i=8;i<15;i++) {
        if( get_bit_value(DataDirection,i) == OUT ) { // An output en-
            abled port
            if( get_bit_value(DataOut,i) == TRUE )
                sbi(PORTA,i-8);
            else
                cbi(PORTA,i-8);
        }
    }

}
```

```
// Return 1 if bit 'bit' of 'data' is set. Else return 0
int get_bit_value( u16 data, int bit)
{

//    u16 mask = 0x1 << bit;  // FIXME. Does not go through Thrint.
    u16 mask = 0x1;

    if( data & mask )
        return 1;
    else
       return 0;
}
/*
// Set bit 'bit' of 'data'.
void set_bit_value(u16 *data, int bit)
{
    //    u16 mask = 0x1 << bit;  // FIXME. Does not go through Thrint.
    u16 mask = 0x1;

    *data = *data | mask;
}

// Unset bit 'bit' of 'data'.
void unset_bit_value(u16 *data, int bit)
{
    //    u16 mask = 0x1 << bit;  // FIXME. Does not go through Thrint.
    u16 mask = 0x1;

    mask  = ~mask;

    *data = *data & mask;
}
*/
//-------------------------------------------------------
//  2. PWM Control
//-------------------------------------------------------

extern inline void init_pwm() // Want this inlined always
{
    outp(0,TCCR0); // Stop both the timers
    outp(0,TCCR2);
}

void enable_pwm(u08 pwm, u08 dutycycle, u08 prescaler )
{
    u08 timer_mask = 0x60;

    timer_mask |= prescaler;

    if( pwm == PWM_0 ) {
        outp(dutycycle,PWM_0_OCR);
        outp(timer_mask,PWM_0);
    }
    else {
        outp(dutycycle,PWM_1_OCR);
        outp(timer_mask,PWM_1);
    }
```

```
}

void disable_pwm(u08 pwm)
{
    outp(0,pwm);
}


//-------------------------------------------------------
//  3. ADC Control
//-------------------------------------------------------

extern inline void init_adc() // Want this inlined always
{
    sbi(ADCSR,7); // set ADEN bit
    sbi(ADCSR,0); // set prescaler to 32
    cbi(ADCSR,1);
    sbi(ADCSR,2);
    cbi(ADCSR,3); // Clear ADIE bit
    cbi(ADCSR,5);
}

extern inline void adc_setchannel( u08 chan )   //  Want  this  inlined
            always
{
    outp( chan,ADMUX);
}


extern inline void adc_enable() // Want this inlined always
{
    u16 dummy;

    sbi(ADCSR, 7); // Set ADEN bit

    // Do a dummy conversion here as per manual guidelines
    dummy = adc_convert();
}

extern inline void adc_disable() // Want this inlined always
{
    cbi(ADCSR,7);  // Clear ADEN bit
    cbi(ADCSR,3);
}

extern  inline  void  adc_set_prescaler(u08  prescaler)  //  Want  this
            inlined always
{
    u08 adcsr_mask;
    adcsr_mask = inp(ADCSR);
    adcsr_mask &= 0xf8; // Zero the last 3 bits
    adcsr_mask |= prescaler; // Now set the prescaler value
    outp(adcsr_mask,ADCSR); // Now update !
}

extern inline u16 adc_convert() // Want this inlined always
{
```

```
    bool done = FALSE;
    u08 status;
    u16 cl, ch,ret;

    sbi(ADCSR,6); // Start conversion

    while( !done ) {
        status = inp(ADCSR);
        if( (status & 0x40) == 0 ) { // Conversion complete
            done = TRUE;
            cl = inp(ADCL);
            ch = inp(ADCH);
            ch <<= 8;
            ret = cl | ch;
        }
    }
}



//-----------------------------------------------------------
//  4. Fan Speed Control
//-----------------------------------------------------------

extern inline void init_fan() // Want this inlined always
{
    cbi(PORTC,FS2);
    cbi(PORTC,FS1);
    cbi(PORTC,FS0);
    sbi(PORTC,SHDN); // Clear it all
    PrevFanSpeed = 0;
}

void shutdown_fan()
{
    unsigned long i;
    bool flag;

    sbi(PORTC,SHDN);

    // Now delay for approx .5 sec ( After integration, this delay
            could be more than .5. But that is not a problem)
    i = 0;
    flag = TRUE;
    while(flag) {
        i++;
        flag = i<0x1E8480;
    }

    cbi(PORTC,FS2);
    cbi(PORTC,FS1);
    cbi(PORTC,FS0);

}

void set_fan_speed(u08 speed)
{
```

```
unsigned long i;
bool flag;

// Avoiding "switch" constructs to satisfy Thrint

if( speed == 0 ) {
    // Follow proper shutdown procedure

    PrevFanSpeed = 0;
    return;
}

cbi(PORTC,SHDN); // Enable fan
if( PrevFanSpeed == 0 ) {
    sbi(PORTC,FS2);
    sbi(PORTC,FS1);
    sbi(PORTC,FS0);

    // Now delay for approx .5 sec ( After integration, this delay
        could be more than .5. But that is not a problem)
    i = 0;
    flag = TRUE;
    while(flag) {
        i++;
        flag = i < 0x3D0900;
    }
}

if( speed == 1 ) {
    cbi(PORTC,FS2);
    cbi(PORTC,FS1);
    sbi(PORTC,FS0);
}
else if( speed == 2 ) {
    cbi(PORTC,FS2);
    sbi(PORTC,FS1);
    cbi(PORTC,FS0);
}
else if( speed == 3 ) {
    cbi(PORTC,FS2);
    sbi(PORTC,FS1);
    sbi(PORTC,FS0);
}
else if( speed == 4 ) {
    sbi(PORTC,FS2);
    cbi(PORTC,FS1);
    cbi(PORTC,FS0);
}
else if( speed == 5 ) {
    sbi(PORTC,FS2);
    cbi(PORTC,FS1);
    sbi(PORTC,FS0);
}
else if( speed == 6 ) {
    sbi(PORTC,FS2);
    sbi(PORTC,FS1);
    cbi(PORTC,FS0);
```

```
    }
    else if( speed == 7 ) {
        sbi(PORTC,FS2);
        sbi(PORTC,FS1);
        sbi(PORTC,FS0);
    }

   PrevFanSpeed = speed;

}



//---------------------------------------------------------------------
//  5. Power Mode Control
//     Controls power mode changes. Assumption is that, device would be
//     woken up again by external events/interrupts.
//---------------------------------------------------------------------

extern inline void init_power_mode()  // Want this inlined always
{

    sbi(MCUCR,SE_BIT); // Enable sleep mode
}

void change_power_mode(int mode)
{

    // TBD. Take care of other pending operations, shutting down other
            units etc, timers/ADC.

    if(mode == SLEEP_MODE_IDLE ) {
        cbi(MCUCR,SM0_BIT);
        cbi(MCUCR,SM1_BIT);
    }
    if(mode == SLEEP_MODE_POWER_DOWN) {
        cbi(MCUCR,SM0_BIT);
        sbi(MCUCR,SM1_BIT);
    }
    if( mode == SLEEP_MODE_POWER_SAVE) {
        sbi(MCUCR,SM0_BIT);
        sbi(MCUCR,SM1_BIT);
    }

    __asm__("sleep"); // Execute a sleep instruction.
                      // Re-awakened by an interrupt.
}

//---------------------------------------------------------------------
//  6. Event Capture Control
//    - Keep it simple - Assume that only positive to negative trans
//      tions on chosen input pins are captured.
//---------------------------------------------------------------------
void check_for_events()
{
    u16 old_data_reg;
    u16 mask = Event_Capture_Mask, bit_mask=1;
```

```
    u08 i;

    // Save old data register
    old_data_reg = DataIn;
    // Latch all ports into DataIn
    read_data_register();

    mask = Event_Capture_Mask;

    for(i=0; i<15; i++) {

        if( mask & 0x1 == 1) { // Event Capture enabled for this port.
            Check if an event happened.
            if( ((old_data_reg & bit_mask) > 0) && ((DataIn & bit_mask)
            == 0)) {
                // Then form a message to be transmitted with the event
            capture info.
                cur_send_msg.type = SN_TYPE_EVENT;
                cur_send_msg.func = SN_FUNC_EVENT_REPORT;
                cur_send_msg.params[0] = i; // Report the port on which
            the event occurred.
                if( enqueue(SEND_QUEUE,cur_send_msg) == FALSE ) {
                    send_queue_ovflow = TRUE;
                }
            }
            mask = mask >> 1;
            bit_mask <<= 1;
        }
    }
}
//-----------------------------------------------------------
//  7. Message Scheduling Control
//-----------------------------------------------------------
// TBD


//-------------------------------------------------------------------
// Smart Node - Message Queue Interface
//-------------------------------------------------------------------
bool enqueue(int queue, SN_Msg in)
{

    if( queue == SEND_QUEUE ) {
        if( queue_is_full(SEND_QUEUE) )
            return FALSE;
        send_buf[send_queue_tail] = in;
        send_queue_contains_data = TRUE;
        modulo_increment(&send_queue_tail);
        return TRUE;
    }
    return FALSE; // Disallow enqueue into RECV_QUEUE. Serves no real
            purpose.
}

bool dequeue(int queue, SN_Msg *ret)
{
    if( queue == RECV_QUEUE ) {
```

```
        if( queue_is_empty(RECV_QUEUE) )
            return FALSE;
        *ret = recv_buf[recv_queue_head];
        modulo_increment(&recv_queue_head);
        if( recv_queue_head == recv_queue_tail )
            recv_queue_contains_data = FALSE;
        return TRUE;
    }

    return FALSE; // Disallow dequeue from SEND_QUEUE. Serves no real
            purpose.
}

void modulo_increment(int *index)
{
    if( *index == BUF_MAX-1)
        *index = 0;
    else
        *index++;
}

extern inline bool queue_is_full(int queue) // Want this inlined always
{
    bool ret = FALSE;
    if( queue == SEND_QUEUE ) {
        if(  send_queue_contains_data  ==  TRUE  &&  send_queue_head  ==
            send_queue_tail )
            ret = TRUE;
    }

    else {
        if(  recv_queue_contains_data  ==  TRUE  &&  recv_queue_head  ==
            recv_queue_tail )
            ret = TRUE;
    }

    return ret;

}

extern inline bool queue_is_empty(int queue) // Want this inlined al-
            ways
{
    bool ret = FALSE;
    if( queue == SEND_QUEUE ) {
        if(  send_queue_contains_data  ==  FALSE  &&  send_queue_head  ==
            send_queue_tail )
            ret = TRUE;

    }

    else {
        if(  recv_queue_contains_data  ==  FALSE  &&  recv_queue_head  ==
            recv_queue_tail )
            ret = TRUE;
    }
```

```
    return ret;
}

//-------------------------------------------------------
//  Error Control and Acknowledgement
//-------------------------------------------------------
void handle_error(SN_Msg *recvd, u08 error_params)
{

    SN_Msg err;

    err.type = SN_TYPE_ACK;
    err.func = error_params;
    *((u32*)&err.params[0]) = *((u32*)&recvd->params[0]); // Copy all
            the 16 bytes of parameter data.
    *((u32*)&err.params[4]) = *((u32*)&recvd->params[4]);
    *((u32*)&err.params[8]) = *((u32*)&recvd->params[8]);
    *((u32*)&err.params[12]) = *((u32*)&recvd->params[12]);

    if( enqueue(SEND_QUEUE,err) == FALSE )
        send_queue_ovflow = TRUE;
}



void send_acknowledgement(SN_Msg *recvd, u16 ack_params)
{
    SN_Msg send;

    send.type = SN_TYPE_ACK;
    send.func = SN_FUNC_ACK_OK;

    send.params[0] = recvd->type;
    send.params[1] = recvd->func;
    *((u16*)&send.params[2]) = ack_params;

    if( enqueue(SEND_QUEUE,cur_send_msg) == FALSE )
        send_queue_ovflow = TRUE;
}
```

2. PID Controller Code

```
/*
 *  Pid.c - Sample secondary thread.
 *  PID Controller Imlementation
 *  Last Updated: 04/30/2003. Author: Vasanth Asokan
 *
 */

#define __AVR_ATmega103__
#include <io.h>
#include <sig-avr.h>
#include <interrupt.h>
#include <stdlib.h>
```

```
#define TRUE  1
#define FALSE 0
#define NTAPS  10   // Window of the filtering function


// Random values for gains. Tune these appropriately based on the sys-
           tem.
#define IGAIN  0xffaa;
#define PGAIN  0xffaa;
#define DGAIN  0xffaa;

#define to_fp(t) ((u16)((u16)(t) << 8))  // convert a raw number to 8.8
           FP representation


typedef unsigned short u16;
typedef unsigned int   u32;
typedef unsigned char  u08;


typedef u16 FP;

typedef struct
{
    FP dState;          // Last position input
    FP iState;          // Integrator state
    FP iMax, iMin;      //  Maximum  and  minimum  allowable  integrator
           state

    FP      iGain,      // integral gain
        pGain,          // proportional gain
        dGain;          // derivative gain
} SPid;


// Globals
// Coefficient buffer
extern FP h_buf[NTAPS];  // Initialized elsewhere

// History/Window buffer
extern FP z_buf[NTAPS];  // Initialized elsewhere

// Linearization parameters
extern FP  rllower, rlupper, // Initialized elsewhere
           rulower, ruupper,
           cllower, clupper,
           culower, cuupper;

extern FP  lrangeratio,urangeratio; // Initialized elsewhere


// Definitions

int main_control_loop()
{

    SPid pid_info;
```

```
    FP output = 0xff00;
    FP pure_output, error, control;
    int pstate = 0;
    int i;

    // Initialize H/W as necessary

    // Initialize PID data structure
    pid_info.dState = 0;  // Fix all these with FP values.
    pid_info.iState = 0;
    pid_info.iMax   = to_fp(10);
    pid_info.iMin   = to_fp(-10);
    pid_info.iGain  = IGAIN;
    pid_info.pGain  = PGAIN;
    pid_info.dGain  = DGAIN;


    // Main infinite while loop
    while(1) {
        // Get plant output
        // Now the controlled output is in 'output'. Remove noise by
            applying FIR filter.
        pure_output = fir_circular(output,NTAPS,h_buf,z_buf, &pstate);
        // Now Linearize this output.
        LinearizeFixed(pure_output, &pure_output);
        // Calculate error
        error = fpsub(pure_output, control );
        // Obtain PID output
        control = UpdatePID(&pid_info,error,pure_output);

        // Apply control effort accordingly.
    }
}

FP UpdatePID(SPid * pid, FP error, FP position)
{
    FP pTerm, dTerm, iTerm;

    pTerm = fpmul(pid->pGain,error);
    // calculate the proportional term

    // calculate the integral state with appropriate limiting
    pid->iState = fpadd(pid->iState,error);
    if (pid->iState > pid->iMax)
        pid->iState = pid->iMax;
    else if (pid->iState < pid->iMin)
        pid->iState = pid->iMin;

    iTerm = fpmul(pid->iGain,pid->iState);   // calculate the integral
            term

    dTerm = fpmul(pid->dGain, fpsub(position,pid->dState));
    pid->dState = position;

    return fpadd(pTerm,fpsub(iTerm,dTerm));
}
```

```
// Calculates and returns M*m with Booth's multiplication algorithm.
// 16-bit multiplication

u32 mult(u16 M, u16 m)
{
    u32 ans = m, q0q_1=0, lastbit;
    u16 *A;
    int count = 16;

    A = (u16*)&ans;
    while( count >= 0 ) {
        if( q0q_1 == 1 ) {
            *A = *A + M;
        }
        else if( q0q_1 == 2 ) {
            *A = *A - M ;
        }

        lastbit = ans & 0x1;
        ans = asr(ans); // do an arithmetic shift right of
        lastbit <<= 1;
        q0q_1 >>= 1;
        q0q_1 |= lastbit;
        count--;
    }

    return ans;

}

// Simulate an arithmetic shift right using logical shift right and
// other operations
u32 asr( u32 in )
{
    u32 sign_mask;

    sign_mask = 0x80000000 & in;  // Extract just the sign bit
    in = in >> 1;                 // A logical right shift

    in |= sign_mask;              // Restore the sign bit again.

    return in;
}

//---------------------------------------------------------------------
// fir_circular - Implements the FIR low pass filter workin on a circu-
          lar buffer
//---------------------------------------------------------------------
FP  fir_circular(FP  input,  int  ntaps,  const  FP  h[],  FP  z[],  int
          *p_state)
{
    int ii, state;
    FP accum=0x0;

    state = *p_state;        // copy the filter's state to a local
```

```
    z[state] = input;        // store input at the beginning of the de-
            lay line

    if (++state >= ntaps) {  // incr state and check for wrap
        state = 0;
    }

    // calc FIR and shift data
    accum = 0;
    for (ii = ntaps - 1; ii >= 0; ii--) {
        accum = fpadd(accum,fpmul(h[ii],z[state]));
        if (++state >= ntaps) {     // incr state and check for wrap
            state = 0;
        }
    }

    *p_state = state;               // return new state to caller

    return accum;
}


//-------------------------------------------------------------------
// LinearizeFixed - Linearize the fixed point raw value. Return con-
            verted value in lin_value
//-------------------------------------------------------------------
int LinearizeFixed(FP raw_value, FP *lin_value)
{
    int conversion_ok = TRUE;
    FP sub_return;
    int flag1, flag2;

    // To avoid unstructured code.
    flag1 = raw_value >= rllower;
    flag1 = flag1 && (raw_value <= rlupper);

    flag2 = raw_value > rulower;
    flag2 = flag2 && (raw_value <= ruupper);

    if( flag1) {
        sub_return= fpsub(raw_value,rllower);
        *lin_value = fpmul(sub_return,lrangeratio);
        *lin_value += cllower;
    }
    else if(flag2) {
        sub_return = fpsub(raw_value,rulower);
        *lin_value = fpmul(sub_return,urangeratio);
        *lin_value += culower;
    }
    //else conversion_ok = FALSE;

    if( !conversion_ok )
        *lin_value = 0;

    return conversion_ok;

}
```

```
//-----------------------------------------------
// Fixed point arithmetic routines
//-----------------------------------------------
u16 fpadd(u16 a, u16 b)
{
    return a+b;
}

u16 fpsub(u16 a, u16 b)
{
    return a-b;
}

u16 fpmul( u16 a, u16 b)
{
    u32 c = (u32)((u32) mult(a,b));

    c >>= 8;
    return (u16)(c & 0xffff);

}
```

3. Host Interface Thread Code

```
// Kumar's version of the HI thread.

#define __AVR_ATmega103__
#include <io.h>
#include <sig-avr.h>
#include <interrupt.h>
#include <stdlib.h>

#include "host.h"

#define arrayPosByte(start, index, retPtr) retPtr=  start + index
#define random() 1


// Start global Data

extern short buffer_tx_head,buffer_tx_tail;
extern short buffer_rx_head,buffer_rx_tail;

extern unsigned char tx_buf_len;
extern unsigned char rx_buf_len;

/* the following variable represents the interface register
 * which contains the data sent by the main controller
 */

extern unsigned char in_buffer[64];
extern unsigned char *ucharPtr,*ucharDataPtr;
extern unsigned char buf_index;
extern unsigned char in_buf_index;
```

```
extern short tx_state;
extern unsigned char status_reg;

extern J1850_Transmit_Buffer buffer_tx[16], buffer_rx[16];
extern J1850_Receive_Buffer *buffer_receive;

extern struct J1850_Data_Buffer *tempBufPtr1, *tempBufPtr2;

// End global data

SIGNAL(SIG_INTERRUPT7)       // signal handler for external interrupt
            int0
{
    unsigned char temp_char;
    temp_char = inp(PINB);
    ++buf_index;
    buf_index %= 64;          // since buffer size is 64(hostInt.h), if
            buf_index goes above 64
                              // wrap it to the beginning of the array
            again.
    in_buffer[buf_index] = temp_char;
    outp(temp_char,PINA);
    outp(0xff,DDRE);
    outp(0x00,PINE);
    outp(0x00,DDRE);
}


void main_control_loop()
{
    unsigned char j =0;
    unsigned char i = 0;
    int eternal,temp;
    int retConditionTransmit, lenTemp;

    short tx_index,temp1,temp2;
    port_init();
    buffer_init();

    cnt_interface_fsm();

}


void buffer_init() {
    char i = 0,j=0;
    unsigned char IFR;
    unsigned char len = 0;
    unsigned char decsn;

    for(i=0; i<25;i++) {
        decsn = ((char)random())&0x1;
        in_buffer[in_buf_index++]= decsn;
        if( decsn == 0x0 ) {
            IFR = ((char)random())&0x1;
            in_buffer[in_buf_index++] = IFR;
            len = (unsigned char) random() % 30;
```

```
            in_buffer[in_buf_index++] = len;
            for(j=0;j<len;j++)
                in_buffer[in_buf_index++] = ((char)random())%256;
        }
    }
    in_buf_index = 0;

}

void port_init(void) {

    outp(0x00,DDRB);
    outp(0xff,DDRA);
    outp(0x00,DDRE);
    outp((1<<INT7),EIMSK);
    outp((1<<ISC71)|(1<<ISC70),EICR);    //  FIXME  -  This  was  EICR.
            Changed to get it to compile.
    sei();

}

void transmit_data() {

    char i;
    short tx_index,temp1,temp2;
    char temp;

    temp = receive_data_from_int();


    if(tx_state==0){
        buffer_tx_head++;
        tx_index = ++buffer_tx_tail;
        tx_state = 1;
    }

    if(tx_state==1){
            if(temp) {
            tx_index = --buffer_tx_head;
            }
            else{
            tx_index = ++buffer_tx_tail;
            }

//          if((tx_buf_len != tx_buf_len %10)) { // Unstructured code.
        if((tx_buf_len != tx_buf_len)) {
            buffer_tx_head = START_INDEX;
            buffer_tx_tail = START_INDEX;
            tx_state = 0;
            return;
            }
        tx_state = 1;
    }

    buffer_tx[tx_index].IFR = temp;
    buffer_tx[tx_index].Valid = 1;
    buffer_tx[tx_index].ProtocolID = J1850_PROT;
```

```
    buffer_tx[tx_index].Length = receive_data_from_int();

    for(i=0;i<buffer_tx[tx_index].Length;i++) {
      // Pair of braces were unmatched. I added this brace - Vasanth
        buffer_tx[tx_index].Data[i] = receive_data_from_int();
    }

    tx_buf_len++;
    if(tx_buf_len == BUF_WARNING)
        status_reg |= 0x1;
    else
        status_reg &= 0xfe;


    if((buffer_tx_head > buffer_tx_tail)){
        buffer_tx_head = START_INDEX;
        buffer_tx_tail = START_INDEX;
        tx_state = 0;
    }

    tx_buf_len++;
    tx_buf_len--;  // Dummy. To prevent Thrint from complaining.
}

void receive_buf_init() {

    buffer_rx_head = buffer_tx_head;

}

void receive(void) {

    char i = 0;
    short int temp1,temp2;

    receive_buf_init();   // inlined code below

    if(buffer_rx[buffer_rx_head].Valid) {
        send_data_to_int(buffer_rx[buffer_rx_head].IFR);     // inlined
            function below
        send_data_to_int(buffer_rx[buffer_rx_head].Length);          //
            inlined function below
        for(i=0;i<buffer_rx[buffer_rx_head].Length;i++){
            send_data_to_int(buffer_rx[buffer_rx_head].Data[i]);     //
            inlined function below
        }
        buffer_rx_head++;
    }

    if(buffer_tx_head > buffer_tx_tail) {
        buffer_tx_head = START_INDEX;
        buffer_tx_tail = START_INDEX;
        tx_state = 0;
    }
    if(buffer_rx_head > buffer_rx_tail) {
        buffer_rx_head = START_INDEX;
```

```
        buffer_rx_tail = START_INDEX;
    }
    if(rx_buf_len == BUF_WARNING)
        status_reg |= 0x2;
    else
        status_reg &= 0xfd;

}

unsigned char receive_data_from_int() {

    unsigned char in_val;

    in_val = 0;
    in_val = in_buffer[in_buf_index];
    in_buf_index++;
    in_buf_index=in_buf_index & 0x3F;

    return in_val;

}

void flush_tx_buffer()
{
    buffer_tx_head = START_INDEX;
    buffer_tx_tail = START_INDEX;
    tx_buf_len = 0;
}

void flush_rx_buffer()
{
    buffer_rx_head = START_INDEX;
    buffer_rx_tail = START_INDEX;
    rx_buf_len = 0;
}

void report_status()
{
    send_data_to_int(tx_buf_len);
    send_data_to_int(rx_buf_len);
}

void send_data_to_int(unsigned char data)
{
    outp(data,PORTA);
}

void cnt_interface_fsm()
{
    int eternal,temp,i = 0;


//    Disabled in original hostInt.c
//    for(eternal=0;eternal<10000;eternal++) {

    temp=receive_data_from_int();
```

```
    if(temp==0){
        transmit_data();
    }

    if(temp==1){
        receive();
    }
    if(temp==2){
        flush_tx_buffer();
    }
    if(temp==3){
        flush_rx_buffer();
    }
    if(temp==4){
        report_status();
    }
}
```