A Study of Compiler-Directed Multithreading for Embedded Applications

Anasua Bhowmik
Computer Sciences Department
University of Maryland
College Park, MD 20742
anasua@cs.umd.edu

Manoj Franklin
ECE Department and UMIACS
University of Maryland
College Park, MD 20742
manoj@eng.umd.edu

Quang Trinh
ECE Department
University of Maryland
College Park, MD 20742
trinh@eng.umd.edu

ABSTRACT

Growing demand for high performance in embedded systems is creating new opportunities to leverage techniques such as pipelining and instruction-level parallel processing, which were originally developed for general-purpose processors. In this paper, we investigate the applicability of compiler-directed multithreading in speeding up embedded applications. In particular, we take programs from the Powerstone benchmark suite—a collection of programs from the embedded applications area—and use our compiler-directed multithreading framework to partition them into multiple threads. While performing the partitioning, the compiler not only considers data dependence information, but also considers control independence information and profile-based information on the most likely control flow paths. Our compiler framework is implemented on the SUIF-MachSUIF platform. The average code expansion due to the introduction of thread information is only 0.82%, but, the performance potential is quite substantial. The effect of different criteria on our thread partitioning technique is evaluated using a trace-driven, multithreaded processor simulator. Our measurements indicate that future embedded processors can speed up the execution of sequential programs with low degrees of multithreading.

Keywords

Branch prediction, control dependence, Powerstone benchmarks, profiling, speculative execution, thread-level parallelism (TLP)

1. INTRODUCTION

Improving the execution speed of embedded applications is becoming an important problem. Any serious attempts at solving this problem should carefully consider the trends in technology. In spite of the severe power consumption requirements, the number of transistors in embedded processors has been rising, primarily due to advances in device technology. This ongoing explosion in device technology is complemented by a similar increase in clock speed. This situation is complicated by a constraint that is germane to embedded processors—low power consumption. Designers of embedded processors have been utilizing the increasing transistor budget to incorporate special features that speed up some aspects of embedded computing. But today embedded processors are being used for a wide variety of applications, and embedded processor designers have begun to include features that are traditionally found in general-purpose processors [16].

Recent studies on multithreading confirm that there is significant performance potential in executing a small number of threads in parallel. Furthermore, the use of multiple hardware sequencers or processing elements (to fetch and execute multiple threads)—besides making judicious use of the available transistor budget increase—fits very nicely with the goal of decentralization, which is very important to deal with on-chip wire delays. Using the increased device count to build additional processing elements (PEs) is indeed a very credible option [2] [5] [13] [14]. The primary means of increasing processor performance, besides increasing the clock speed and reducing the memory latency, has always been the exploitation of the inherent parallelism present in programs, with the use of a combination of software and hardware techniques. Although the majority of previous research in embedded processors focused on a single thread of execution, a more effective increase of parallelism can be achieved from the execution of multiple threads belonging to the same program[1].

This paper investigates the potential of software (compiler-based) techniques to partition sequential embedded programs into multiple threads that the hardware can execute in parallel. Because the compiler has an overall view of the program, it can find the control independent points in the program and partition the sequential program into multiple threads. It can also determine data dependences between distant code. We use both of these features, along with profile-based data on likely control flow paths, to partition sequential programs into multiple threads. Thus, our compiler based thread partitioning algorithm takes into account both control and data independence to do effective thread partitioning. From the hardware side, data value prediction is incorporated to reduce the effect of inter-thread data dependences.

Our studies with embedded applications have led to the following observations:

- The performance potential of single-threaded processors is fairly limited.

[1]The term "thread" has different meanings in different contexts; our notion of threads is finer than the coarse-grain OS-level threads, and comprise of tens to hundreds of instructions.

- Compiler-directed speculative multithreading, along with data value speculation, has good potential to speed up embedded applications
- Some embedded programs benefit from the use of non-loop threads

The rest of this paper is organized as follows. Section 2 provides background information on multi-threading for embedded systems applications. Section 3 presents an overview of our multi-threading compiler framework. Section 4 presents an experimental evaluation of the compiler-generated threads for the Powerstone benchmarks. Section 5 presents a summary and the major conclusions of this paper.

2. MULTITHREADING FOR EMBEDDED APPLICATIONS

2.1 Constraints for Embedded Processors

Embedded processors currently form an important sector of the processor market. They are particularly used in many applications in the communications and mobile computing area. Although the basic tenets of computing in the embedded systems world are the same as those in the general-purpose computing world, there are some additional constraints to be considered while designing embedded processors. These constraints concern primarily with power dissipation, code size, and die size. Many embedded processors are used in applications such as cellular phones where the power supply is derived from a battery. For such applications, it is very important that the power consumption of the processor is as low as possible. Many embedded systems are also constrained by memory size and die size limitations. Limited memory size implies that the code size should be as small as possible. In spite of these special constraints for embedded systems, the demands on the processing power for embedded applications has been steadily rising.

2.2 Parallelism in Powerstone Benchmark Programs

It is worthwhile to characterize embedded applications. In particular, we like to know how much parallelism exists, what kind of branch prediction accuracies we can obtain, etc. To that end, we measure the available parallelism (under different machine models) present in the Powerstone benchmarks, a collection of embedded application programs including automobile control, signal processing, graphics and fax applications. A description of the benchmarks is given in Table 1. These portable and embedded benchmarks are used to make design trade-offs in the architecture and the compiler of the Motorola low power M-CORE processor [16]. For this study, we use a software tool called TAPE (Tool for Available Parallelism Estimation) [3]. TAPE performs trace-based simulation, and performs a parallelism *limit study* by constructing a dynamic dependence graph (DDG) based on the different kinds of dependences present among the instructions of the trace. TAPE allows different models for handling control dependences: realistic branch prediction, realistic branch prediction augmented with exploitation of control independences, and perfect branch prediction.

Let us take a quick look at the amount of parallelism available in the Powerstone benchmarks under these different control flow models. Table 2 presents the available parallelism obtained for 3 abstract machine models (given in 3 columns): (i) an execution model in which control speculation is employed within a window of 32 instructions, but control independence is not utilized, (ii) an execution model in which control speculation is employed, and control independence is utilized whenever a branch is mispredicted within a window of 256 instructions, and (iii) an execution model that utilizes perfect branch prediction and a window size of 256 instructions. The first case indicates a limit of what can be achieved by ILP (instruction-level parallelism) techniques, and the second indicates the potential of pursuing multiple threads. These measurements were done with the Alpha instruction set architecture.

Benchmark	Description
auto	Automobile control application
bffo	
bilv	Shift, AND, OR operation
blit	Graphics application
compress	A Unix utility
des	Data Encryption standard
fir_int	
g3fax	Group three fax decode (Single level image decompression)
ucbqsort	U.C.B. Quicksort

Table 1: Powerstone benchmark suite

Table 2 also presents the branch prediction accuracies obtained for the benchmarks. Whereas many of the benchmarks obtain very high prediction accuracies, in the range 96%-99.9%, there are a few that obtain substantially low prediction accuracies—`compress` (91.0%), `ucbqsort` (81.06%), and `des` (75.42%). The parallelism obtained by branch prediction alone is naturally low for these three benchmarks (around 5). The column that is of particular interest to us is the penultimate one, because it shows the potential of multithreading to improve performance. On looking at this column, we can see that except for `auto`, `des`, and `g3fax`, the others can obtain reasonable performance enhancements by small-scale multithreading.

This characterization also indicates that for most of the programs in the Powerstone benchmark suite, instruction-level parallelism (ILP) techniques can capture only a limited amount of parallelism.

2.3 Speculative Multi-threading for Embedded Processors

Many of the embedded applications are non-numeric in nature. In particular, in such applications memory addresses are difficult (if not impossible) to statically predict—in part because they often depend on run-time inputs and behavior—that makes it extremely difficult for the compiler to statically prove whether or not potential threads are independent. To deal with these difficulties, the speculative multithreading (SpMT) model has been found to be more effective [8] [15]. This model is particularly important to deal with the complex control flow present in typical non-numeric programs. In this model, threads are extracted from sequential code and run in parallel, without violating the sequential program semantics. This means that inter-thread com-

Table 2: Available Parallelism with Different Control Flow Models

Benchmark	Branch prediction accuracy	Available parallelism with		
		No utilization of control independence	Utilization of control independence	Perfect branch prediction
auto	99.86%	6.67	6.77	6.77
bffo	99.23%	10.00	20.59	20.60
bilv	97.14%	12.16	15.89	15.89
blit	99.90%	9.99	10.22	10.22
compress	91.00%	5.44	10.25	14.93
des	75.42%	4.44	8.52	9.27
fir_int	97.09%	10.18	19.45	19.56
g3fax	96.09%	5.70	7.51	7.84
ucbqsort	81.06%	5.65	10.78	15.73

munication between any two threads (if any) is strictly in one direction, as dictated by the sequential thread ordering. Thus, no explicit synchronization operations are necessary, as the sequential semantics of the threads guarantee proper synchronization. Program correctness will not be violated if at run time there is a true data dependence between two threads. The purpose of identifying threads in such a model is to indicate that those threads are good candidates for parallel execution.

3. COMPILER BASED THREAD PARTITIONING

In this section we provide a brief description of our compiler framework for thread partitioning. A detailed description is beyond the scope of this paper; the objective of this paper is to study the effectiveness of multithreading for embedded applications.

3.1 Multi-threaded Architectural Model

The multi-threaded architectural model assumes that the program has been partitioned into a collection of threads. Each thread can spawn any arbitrary number of threads. A particular thread can also be spawned from different places. Threads can be spawned speculatively if required; i.e., a thread can be spawned before knowing for sure that control flow will reach that thread. If it is found that the control speculation was wrong, then the speculative thread is squashed from its PE. But other threads spawned by this speculative thread will be aborted, only if those threads are also control dependent on the same branch. If they are control independent of that branch they can continue execution.

In its general form, this multi-threaded processor hardware consists of a number of processing elements (PEs). Each PE has its own program counter, fetch unit, decode unit, and execution unit, so as to fetch and execute instructions from the thread currently assigned to it. The PEs are connected together by an interconnection network.

3.2 Compiler Framework

In this subsection we briefly describe our compiler framework for thread partitioning. The layout of our overall system is shown in Figure 1.

While partitioning the program into threads, the compiler has to consider three mutually independent factors—*data dependence*, *control dependence*, and *thread size*—together, to decide a good partitioning. Partitioning programs into threads for non strict languages (like C) such that total execution time is minimized, is an NP-Complete problem. So we formulate some metrics and use them to find a good solution of the partitioning problem.

In the following subsections we discuss how the compiler takes care of data dependence, control dependence, and the thread size. The compiler does the program analysis and partitioning on a high level intermediate representation. The high level representation retains all the source level pointer and type information, and hence it is possible to take into account the dependences due to pointer aliasing and array references. Hence the compiler is able to extract parallelism even from pointer intensive programs. We assume that the multi-threaded architecture can take care of the anti- and output- register dependences with dynamic register renaming. We have used the profiling information to find out the most likely path, that the control will take and this information is used by the compiler to specify threads that are to be spawned speculatively.

3.3 Program Profiling

We have used a separate compiler pass to instrument the source code and to gather the profiling information. In the profiling pass, we find out for every basic block, which basic block is most likely to be visited next. The compiler uses this to find out the most likely path and also to estimate the number of instructions that would be executed between two basic blocks. Furthermore, we find out the number of loop iterations using the profiling information. The estimate on the number of loop iterations helps us to decide whether to execute the loop iterations in parallel even in presence of available parallelism. We will discuss this in details in the following subsections.

3.4 Data Dependence

In our framework we formulated a metric called *data dependence count* to partition the programs such that the data dependence between threads are minimized.

Our thread partitioning algorithm works in multiple passes. In the first pass, the compiler builds the CFG and also finds out the data dependence information. It does the traditional data flow analysis and calculates the *read/write* sets [1] for every instruction. We have implemented an intraprocedural pointer analysis to have an improved data dependence information. The pointer analysis helps us in getting more

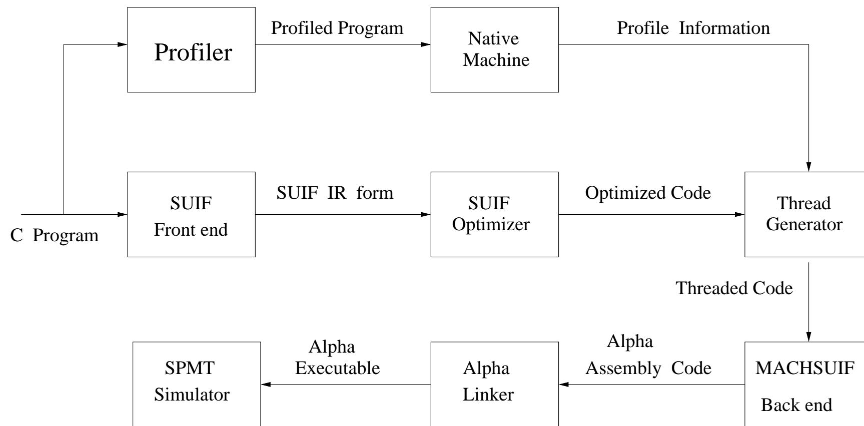

Figure 1: The Layout of the Compiler and Simulator Framework

precise read/write sets. After calculating the *read/write* sets for every instruction, data flow analysis is performed and for every variable in the read set of an instruction, the set of reaching definitions [1] are determined.

The *data dependence count* (DDC) is the weighted count of the number of data dependence arcs coming into a basic block from other blocks as shown in Figure 2. This models the extent of data dependences this block has on other blocks. If the dependence count is small, then this block is more or less data independent from other blocks and it may be beneficial to begin a thread at the beginning of that basic block. While counting the data dependence arcs, the compiler gives more weights to the arcs coming from blocks that belong to the threads closer to the block under consideration. The dependences from distant threads are likely to be resolved earlier and hence the current thread is less likely to wait for the data generated in that thread. Moreover, we give less weightage to the data dependence arcs coming from the less likely paths. The advantage of using this metric are twofold. First of all, it is much simple to compute. Also we found it more accurate than other sequential execution based modeling in the presence of out-of-order execution inside each thread.

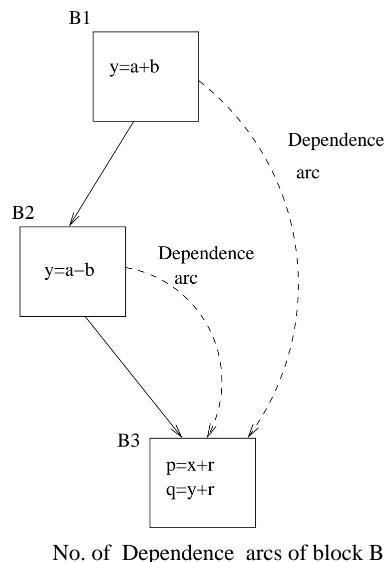

Figure 2: The Data Dependence Arcs

3.5 Program Partitioning

This subsection describes the partitioning algorithm. The compiler partitions the CFG into multiple threads, and also specifies the points in the program from which a particular thread can be initiated. In the partitioning algorithm, I have used the basic blocks as the granularity of partitioning, i.e., either all the instructions inside a basic block are included in a thread or none of them are included. In other words, we do not split a basic block across multiple threads. From every basic block, A, the compiler looks ahead until the basic block B, which is control independent of A and decides which future threads could be initiated from A. Also, at this point the compiler decides which basic blocks in the path between A and B can be included in the current thread, i.e. the thread containing A. To maintain load balancing between the threads, it uses a lower limit and an upper limit for the number of instructions that can be executed in one thread. It also selects speculative threads based upon the profiling data. It selects the most likely path that the program will take for going from basic block A to its next control independent point B. The compiler partitions the program such that the execution in the most likely path be optimized. The thread will continue execution in the speculated path and if it finds the speculation to be incorrect at later point, it will take the correct path. However, there is no need to abort the threads that are spawned at the control independent point of this thread.

Several cases may arise when we look inside the most likely path between the basic blocks A and B. These are shown in Figure 3. The likely path between the basic blocks A and B are shown by thick arrow. In Figure 3(a), basic blocks A and B are not very far and also by including the instructions executed in the likely path between A and B, (including B) in thread 1, the size of thread 1 is not going to violate the upper limit. So the compiler does not spawn a new thread at B. Rather the compiler includes all blocks between A and B in thread 1 and looks beyond B to find the next potential thread starting point. In figure 3 (b), B is not too close to A and yet not too far from A. So B is a potential thread starting point. So the compiler marks B as the starting

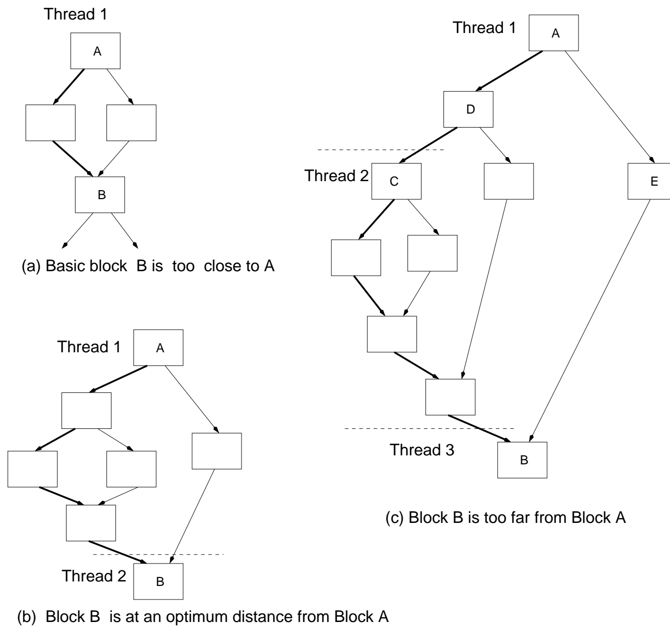

Figure 3: Different Cases in Program Partitioning

point of thread 2 and forms a thread at B. Now it checks the data dependence between the thread containing A and the thread containing B according to the data dependence distance or the data dependence count. If it is found that the total completion time of threads 1 and 2 (where thread 2 is spawned from the beginning of B), is less than the completion time if the two threads are executed sequentially one after another, then it spawns thread 2 from A.

In figure 3(c), A and B are very far apart, as far as the most likely path between them are concerned. So, starting a new thread at B and including all the blocks till B in thread 1 is not efficient. First of all, the size of thread 1 will become very large and moreover there may exist potential threads inside the likely path between A and B. So the compiler looks inside the likely path between A and B and tries to partition it further. In case of 3(c) it is found that basic block C is a starting point of thread 2 and this thread is speculatively spawned from A, before the actual direction of the branch is resolved. Thread 3, which starts from basic block B is spawned from somewhere inside thread 2.

The compiler also checks the paths that are not the likely paths and partitions them as well. If at run-time, control goes into those unlikely paths, then the threads spawned speculatively are aborted. But the threads that are not control dependent on the aborted threads need not be aborted. For example, consider Figure 3 (c). If from A, instead of following the most likely path, the control goes to basic block D, when both threads 2 and and 3 have been spawned, thread 2, would be aborted, but not thread 3, as B is control independent of A. Moreover in the path containing D, there can be spawning of thread 3 as well, and this spawning should be ignored during execution because thread 3 has already been spawned.

In our compiler framework, the loops are treated as a special case of control dependence. For loops the compiler checks the dependence between two iterations of the loops, and if it is found that spawning another thread for the next iteration is profitable, then the thread is spawned. It may also happen that, instead of spawning from the beginning of the loop for the next iteration, the compiler spawn the next iteration from somewhere inside the loop. The large body of the loops may be further partitioned into multiple threads as described above. While partitioning the loops, we use profile information on the number of loop iterations. Typically the compiler does not want to execute small loop body in parallel. However, if the number of iterations is large then the compiler would spawn the iterations as separate threads. Otherwise the size of the thread will become very large.

3.6 Implementation in the SUIF Platform

Our thread partitioning algorithm has been implemented on the SUIF-MachSUIF platform [9]. All of the compiler analysis and thread partitioning are done at the high-level intermediate representation (IR) of SUIF. We have chosen the SUIF platform to implement our compiler system because it provides a modular and flexible infrastructure to develop compiler optimizations. SUIF first translates high-level source code into an IR, and then performs code optimization through several independent passes on that IR. While transforming high-level programs into IR, SUIF retains all of the relevant information from the high level source program. This is particularly helpful for carrying out optimization such as profiling and pointer analysis. Moreover, the instructions in the SUIF IR are very close to the assembly level instructions; thus, the estimation of thread sizes done at the IR level remains valid in the final assembly level as well. In SUIF, it is possible to annotate the instructions with necessary information like data dependence, and use them in separate passes afterwards. Also, the SUIF package contains many optimization modules, which improve the quality of the code produced. We have used the MachSUIF [17] framework to generate Alpha assembly code from the SUIF IR.

4. EXPERIMENTAL STUDY

In order to see how much of the parallelism measured in Section 2.2 can potentially be tapped by our compiler-directed multithreading approach, we enhanced our software tool (TAPE) along the lines of the simulation environment used in [15] to study parallelism in general-purpose applications. The number of PEs, issue size per PE, etc., are parameterized. It models a perfect instruction cache and data cache. The code executed in the supervisor mode are unavailable to the simulator, and are therefore not taken into account in the measurements. Furthermore, the simulator does not overlap the execution of threads that precede and succeed a system call. For these measurements, each PE has an issue width of 4 instructions per cycle, an instruction window of 32 entries, and can perform out-of-order execution.

When encountering a conditional branch instruction in a thread, its PE consults a branch predictor for a prediction. If the prediction is incorrect, the immediate control-independent point in the program is determined. If this point is within the thread, then subsequent threads are not squashed. Branch predictions are done using a 2-level Pap scheme [20], with a direct-mapped 16K-entry Branch History Table, a pattern size of 6, and 3-bit saturating counters in the Pattern History Table entries.

A data value predictor is implemented in the simulator. This predictor is a hybrid of a stride predictor and a 2-level predictor [19]. The first level of the predictor has 16K entries, and is direct-mapped. Data value prediction is carried out only for those instructions that produce a single register result. Thus branch instructions, store instructions, nops, and double-precision instructions are not considered for data value prediction.

4.1 Code Explosion Due to Thread Information

As mentioned earlier, for embedded system applications, it is important that our thread partitioning does not have a significant impact on the code size. Table 2 gives the increase in code size because of including thread information in the program binary. For each benchmark program, the table provides the number of static instructions for Alpha ISA, the number of instructions with annotation, and the percentage of instructions with annotation. From the table, we can see that the code size expansion ranges from 0.12% to 2.69%, with an average of 0.82%, which is very insignificant. The number of instructions that are annotated does not depend on the ISA; rather it depends on the program characteristics. For annotating an instruction, we need 16/32 bits to specify an instruction address and 2 bits to specify the type of annotation. Note that for embedded processors where memory size is usually smaller, the number of bits required to specify the instruction address will be even less. Also, we can use special hardware and have relative addressing scheme to specify the instruction address in the annotations, thereby reducing the space requirements further.

Benchmark	Static Instruction Count	Additional Instrs for Conveying Thread Info	Code Expansion Factor
auto	1359	5	0.37%
bffo	1339	6	0.45%
bilv	1671	2	0.12%
blit	1495	8	0.54%
compress	2190	59	2.69%
des	2007	8	0.40%
g3fax	1561	17	1.09%
ucbqsort	1787	16	0.90%
Average	1676.12	15.12	0.82%

Table 3: Increase in code size of the benchmark programs because of including thread information

4.2 Parallelism Without Data Value Prediction

Our first set of multithreading studies were done without employing data value prediction. Figure 4 presents the overall parallelism in terms of instructions per cycle (IPC) obtained in these experiments, with different number of processing elements (PEs). All benchmarks are simulated till hundred million instructions unless the programs get completed before that.

From Figure 2, we can see that across all benchmark programs, there is notable speedup with 4 PEs, except for `bilv`, `blit`, `g3fax`, and `ucbqsort`. Among these, `g3fax` did

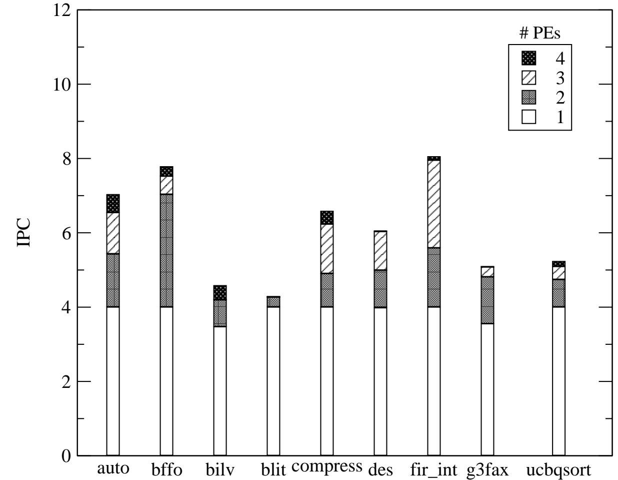

Figure 4: Parallelism Results for varying number of PEs when Data Value Prediction is not Employed

not show any parallelism in the measurements in Section 2. Even for the ones that show promise, the parallelism saturates when the number of PEs reaches 2 or 3. Thus, without data value prediction, multithreading has limited use for these embedded applications.

4.3 Parallelism With Data Value Prediction

Next, we present the parallelism values obtained when data value prediction was employed in the multithreaded processor. Figure 5 presents the parallelism values obtained in these experiments, with data value prediction. For ease of comparison, the results from Figure 2 are reproduced alongside.

When data value prediction is employed, two of the four benchmarks that did not show much parallelism—`blit` and `ucbqsort`—show a marked improvement. In addition, `bffo` shows further improvement. The most notable speedup is seen for `blit`. On the other extreme, `bilv` does not show any noticeable speedup with multi-threading, even when data value prediction is employed. Most programs have substantial speedups with multithreading. By and large, incorporating data value prediction helps to reduce the effects of inter-thread data dependences, thereby providing notable speedups. Thus, we can see that multithreading is quite effective for embedded systems programs when the processor employs data value prediction, which is quite encouraging.

4.4 Importance of Non-loop Threads

The experimental measurements conducted so far included threads that are loop-centric (iterations of loops) as well as non-loops. In the next set of measurements, we measure the parallelism obtained when only loop-centric threads are employed. Figure 6 presents these results. For each benchmark, two bars are given. The first corresponds to using all kind of threads, and the second corresponds to using only loop-centric threads. Among the benchmarks, only a single program—`bffo`—benefits from using only loop-centric threads. For most of the programs, restricting to loop-centric threads results in less parallelism being exploited. This demonstrates the importance of a multithreading frame-

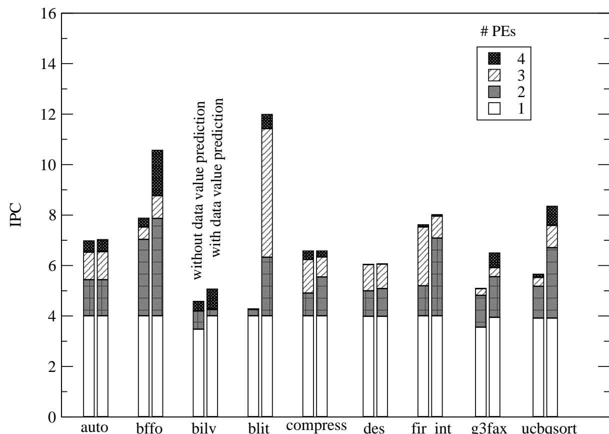

Figure 5: Parallelism Results for varying number of PEs when Data Value Prediction is Employed

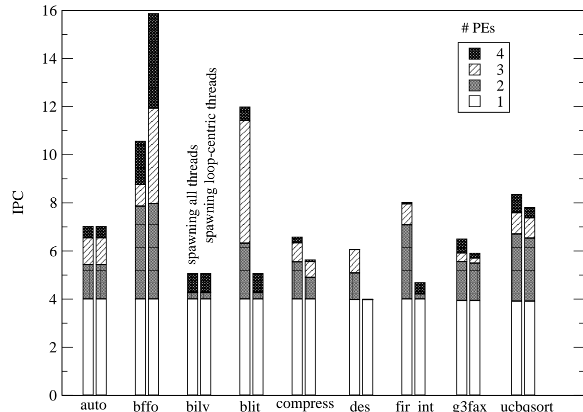

Figure 6: Parallelism Results for varying number of PEs when Using Different Types of Threads

work that supports loop-centric as well as other kind of threads.

4.5 Effect of Selective Loop Unrolling

Some of the embedded applications are dominated by very small loops, whose iterations are fairly independent. If each iteration of such a loop is partitioned as a single thread, then each thread becomes very small, and the thread overhead becomes too much. On the other hand, if the entire loop is made a single thread, then we fail to exploit the inter-iteration parallelism present in these loops. In order to deal with this problem, we experimented with selective unrolling of loops. Selective unrolling is very important to keep the code expansion factor low. We introduced a selective unrolling pass in our compiler framework; currently this pass unrolls simple "`for`" loops with fixed upper and lower bounds. The use of selective unrolling showed substantial potential for 4 of the benchmarks—`auto`, `compress`, `fir_int`, and `ucbqsort`. The code expansion due to selective loop unrolling for these 4 benchmarks were 58.8%, 15.1%, 33.11%, and 7.0%, respectively. The improvements in thread-level parallelism are shown in Figure 5. For each of these 4 benchmarks, 2 bars are shown, the first indicating the parallelism before loop unrolling and the second indicating the parallelism after loop unrolling. Among these two benchmarks, `auto` and `fir_int` show remarkable improvement due to selective loop unrolling.

5. DISCUSSION AND CONCLUSIONS

Embedded processor designs are constrained by power consumption, code size, and die size limitations. Nevertheless, the performance expected from them has been steadily increasing. Today's embedded processors incorporate a variety of techniques that are used for general-purpose processor design.

To obtain high performance in embedded system applications, it is important to handle both loop-terminating branches and other conditional branches in an efficient manner. Although traditional branch prediction provides somewhat reasonable prediction accuracies for embedded system application programs, a substantial increase in performance for next-generation systems requires more effective ways of dealing with conditional branches. Recognizing control-independent regions, and performing speculative execution along multiple (independent) flows of control have the potential to extract the large amounts of parallelism that are available at a distance. Given the increasing interest in multithreading for general-purpose processors, we expect that future embedded processors will also attempt to execute multiple threads in one way or another.

This paper investigated the applicability of multithreading in embedded system applications. We partitioned programs from the Powerstone benchmark suite, a collection of programs from the embedded systems area, into multiple threads. While performing the partitioning, the compiler not only considers control independence information, but also considers data dependence information and profile-based information on the most likely control flow paths. We performed several measurements with these compiler-generated threads. Our measurements show that a majority of the benchmarks programs are able to get substantial increase in parallelism when up to 4 threads are executed in parallel, provided data value speculation is used to break inter-thread data dependences. Our measurements also show that most of the benchmark programs require non-loop threads also, in addition to loop-centric threads. Results from these simulations indicate that future processors can speed up the execution of embedded system program by using multithreading.

A major advantage of speculative multithreading multithreading is backward compatibility with existing processors. That is, an existing executable program for the original (single-threaded) embedded processor forms legal single-thread code for a multithreading embedded processor. This feature is very important from the commercial point of view, because of customers' strong preference to have the ability to run the old binaries in the new machine (although those binaries can not benefit from the new machine's multithreading features).

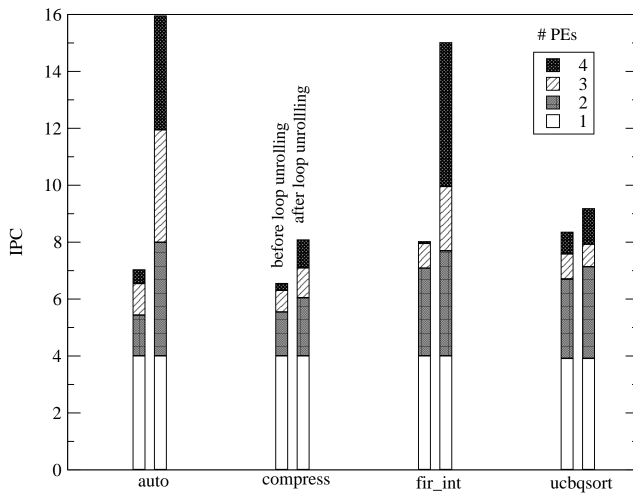

Figure 7: Parallelism Results for varying number of PEs when Selective Loop Unrolling is Employed

Acknowledgements

This work was supported by the U.S. National Science Foundation (NSF) through a CAREER grant (MIP 9702569), a regular grant (CCR 0073582), and an REU grant (EIA 9912218).

6. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] H. E. Bal and M. Haines, "Approaches for Integrating Task and Data Parallelism," *IEEE Concurrency*, July-September 1998.

[3] A. Bhowmik and M. Franklin, "A Characterization of Control Independence in Programs," *Proceedings of Workshop on Workload Characterization*, 1999.

[4] R. Cytron, J Ferrante, B. Rosen, M. Wegman, and F. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 4, pp. 451-490, October 1991.

[5] W. J. Dally and S. Lacy, "VLSI Architecture: Past, Present, and Future," *Proceedings of Advanced Research in VLSI Conference*, 1999.

[6] P. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton, "Single-Program Speculative Multithreading (SPSM) Architecture: Compiler-assisted Fine-Grained Multithreading," *Proc. International Conference on Parallel Architecture and Compilation Techniques (PACT '95)*, 1995.

[7] M. Emami, "A Practical Interprocedural Alias Analysis For an Optimizing/Parallelizing C Compiler," *Masters Thesis*, School of Computer Science. McGill University, Montreal, 1993.

[8] M. Franklin, "The Multiscalar Architecture," *Ph.D. Thesis, Technical Report 1196*, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[9] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. W. Liao, E. Bugnion, and M. S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer*, December 1996.

[10] P. Y. T. Hsu and E. S. Davidson, "Highly Concurrent Scalar Processing," *Proc. 13th Annual International Symposium on Computer Architecture*, pp. 386-395, 1986.

[11] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," *Proc. 19th Annual International Symposium on Computer Architecture*, pp. 46-57, 1992.

[12] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, "Characterizing the Impact of Predicated Execution on Branch Prediction," *Proc. 27th International Symposium on Microarchitecture*, pp. 217-227, 1994.

[13] O. C. Maquelin, H. H. J. Hum, and G. R. Gao, "Costs and Benefits of Multithreading with Off-the-Shelf RISC Processors," *Proceedings of the First International EURO-PAR Conference* (Seif Haridi, Khayri Ali, and Peter Magnusson, eds.), no. 966 in Lecture Notes in Computer Science, Stockholm, Sweden, pp. 117-128, Springer-Verlag, August 29-31, 1995.

[14] K. Olukotun, et al, "A Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Transactions on Computers*, September 1999.

[15] J. Oplinger and D. Heine and M. S. Lam. In Search of Speculative Thread-Level Parallelism. *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1989.

[16] J. Scott, L. H. Lee, A. Chin, J. Arends, and B. Moyer, "Designing the MCORE M3 CPU Architecture," *Proc. International Conference on Computer Design (ICCD)*, 1999.

[17] M. D. Smith, G. Holloway, "An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization".

[18] K. B. Theobald, G. R. Gao, and L. J. Hendren, "On the Limits of Program Parallelism and its Smoothability," *Proc. 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pp. 10-19, 1992.

[19] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors", *Proceedings of 13th IEEE/ACM International Symposium on Microarchitecture*, pp. 281-90, Dec. 1997.

[20] T-Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proc. 19th Annual International Symposium on Computer Architecture*, pp. 124-134, 1992.