# A Feasibility Study of Hierarchical Multithreading

Mohamed M. Zahran
ECE Department
University of Maryland
College Park, MD 20742, USA
mzahran@eng.umd.edu

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

## Abstract

*Many studies have shown that significant amounts of parallelism exist at different granularities. Execution models such as superscalar and VLIW exploit parallelism from a single thread. Multithreaded processors make a step towards exploiting parallelism from different threads, but are not geared to exploit parallelism at different granularities (fine and medium grain). In this paper we present a feasibility study of a new execution model for exploiting both adjacent and distant parallelism in the dynamic instruction stream. Our model, called* hierarchical multithreading, *uses a two-level hierarchical arrangement of processing elements. The lower level of the hierarchy exploits instruction-level parallelism and fine-grain thread-level parallelism, whereas the upper level exploits more distant parallelism. Detailed simulation studies with a cycle-accurate simulator are presented, showing the feasibility of hierarchical multithreading. Conclusions are drawn about the best ways to obtain the most from the hierarchical multithreading scheme.*

## 1. Introduction

A defining challenge for research in computer science and engineering has been the ongoing quest for faster execution of programs. There is broad consensus that barring the use of radically novel technologies such as quantum computing and biological computing, the key to further progress in this quest is to do parallel processing at different granularities. Many studies [7][9][13] have confirmed that a lot of parallelism exists at different granularities.

The commodity microprocessor industry has been traditionally looking to fine-grained or instruction-level parallelism (ILP) for improving performance, by means of sophisticated microarchitectural techniques and compiler optimizations. But exploiting parallelism at this fine granularity and from a single thread seems to reach its limit, and the need for exploiting parallelism at different granularities and from multiple threads arises. Many proposals such as multiscalar [4][11], trace processing [10], superthreading [12], and clustered multithreading [3] have been proposed to exploit thread-level parallelism (TLP). However, all of these proposals exploit TLP at a single granularity only.

In this paper we investigate the potential of a hierarchical multithreading model to exploit TLP at two granularities. It makes use of decentralization to overcome the effects of increasing wire delays. We explore the feasibility of this execution model to be a candidate for next generation processors, and investigate issues that need to be addressed for extracting good performance.

The rest of the paper is organized as follows. Section 2 presents the background and related work. Section 3 describes the HMT (Hierarchical Multithreading) model. Section 4 describes the HMT microarchitecture that we use in our evaluation. Section 5 presents the experiments we conducted, along with some analysis. Finally we conclude in section 6.

## 2. Background and Related Work

Limited size instruction window is one of the major hurdles in exploiting parallelism. Programs typically have millions of dynamic instructions, whereas the window size is only two or three dozens of instructions. In order to extract lots of parallelism, we need to have a large instruction window, which has the effect of increasing the run-time visibility into the program structure. However, having a very large instruction window is difficult, for the following reasons: (1)Implementation constraints limit the window size. (2)Large instruction windows increase the complexity of the instruction scheduler, thus increasing the cycle time. (3)Branch misprediction reduces the number of useful instructions in the instruction window.

Programs tend to have parallelism at different granularities. ILP refers to parallelism present among instructions belonging to the same thread, while TLP (thread-level parallelism) refers to parallelism present across instructions of different threads. The amount of ILP present in a program may not be high enough to obtain high performance. The multithreading idea aims at extracting TLP.

### 2.1. Speculative Multithreading (SpMT)

The central idea behind multithreading is to have multiple flows of control within a process, allowing parts of the process to be executed in parallel. In the traditional *parallel threads* model [2], threads that execute in parallel are control-independent, and the decision to execute a thread does not depend on the other active threads. However, under this model, compilers and programmers have had little success in parallelizing highly irregular numeric applications and most of the non-numeric applications. For such applications, researchers have proposed a different thread control flow model called **sequential threads** model, where threads are extracted from sequential code and run in parallel, without violating the sequential program semantics. Inter-thread communication between two threads (if any) will be strictly in one direction, as dictated by the sequential thread ordering. No explicit synchronization operations are necessary. The purpose of identifying threads in such a model is to indicate that those threads are good candidates for parallel execution in a multithreaded processor.

Prior proposals using sequential threads are the multiscalar model [4][11], the superthreading model [12], the trace processing model [10], and the dynamic multithreading model [1]. In the sequential threads model, threads can be *nonspeculative* or *speculative* from the control point of view. A thread is non-speculative if it is guaranteed to commit, and is speculative otherwise. If a model supports speculative threads, then it is called **speculative multithreading (SpMT)**. This model is particularly useful to deal with the complex control flow present in typical non-numeric programs. In fact, many of the prior proposals using sequential threads implement SpMT [4][6][8][10][11][12].

### 2.2. Limitations of Single-Level Multithreading

The speculative multithreading architectures proposed so far use a single level of multithreading. The program is partitioned into threads, and multiple threads are run in parallel using multiple PEs. The PEs are usually organized as a circular queue in order to maintain sequential thread ordering, as indicated in Figure 1. A major drawback associated with single-level multithreading is that it is limited to exploiting TLP at one granularity only, namely the size of each thread (although it can exploit ILP within each thread). Thus, if it exploits fine-grain TLP, then it does not exploit more coarse-grain TLP, and vice versa. In order to obtain high performance, we need to extract parallelism at different granularities.

A second drawback of single-level multithreading processors is that they cannot easily exploit control independence between multiple threads. If we try to modify the hardware associated with the circular queue in order to take advantage of control independence, the design becomes too complicated.

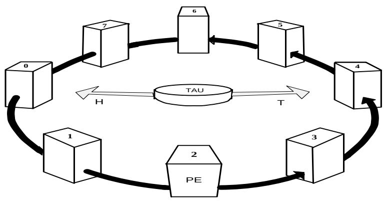

**Figure 1. Circular Queue Arrangement of PEs in a Multiscalar Processor**

Another factor that affects the scalability of single-level multithreading is thread prediction accuracy. If the thread prediction accuracy is 98% for the first PE, then it will drop to around 90% for the 5th PE, and to around 80% for the 10th PE. This means that the payoff as we increase the number of PEs will drop exponentially.

Finally as we increase the number of PEs, the maximum distance between two PEs increases substantially. This wire delay will be a severe bottleneck.

## 3. The HMT Thread Model

We investigate *hierarchical multithreading (HMT)* to overcome the limitations of single-level multithreading. An important attribute of any multithreading system is its thread model, which specifies the sequencing of threads, the names (such as registers and memory addresses) threads can access, and the ordering semantics among these operations, particularly those done by distinct threads. First, we will discuss HMT's thread sequencing model, which specifies ordering constraints (if any) on multiple threads. Then, we discuss HMT's inter-thread communication model, which deals with passing data values among two or more threads.

### 3.1. Multithreading Vs HMT

Figure 2 shows the differences between single threading, multithreading and hierarchical multithreading. In single threading, instructions are fetched from a single thread. Depending on the fetch capacity, the instruction window, and the issue width, fine-grain parallelism is exploited. The processor is not aware of the presence of independent instructions outside of this window. In multithreading, several instructions from several threads are fetched and executed, thus taking advantage of independent instructions from several threads. But in this scheme threads may be nearby in the dynamic instruction stream, thus the processor is not exploiting parallelism from threads far away in the program. In hierarchical multithreading, threads (will be called *tasks* from this point on) far apart in the instruction stream are executed in parallel through partitioning the program first into supertasks, as indicated in the next section, thus taking advantage of instruction- and thread-level parallelism as well as control independence between far apart threads.

### 3.2. Thread Sequencing Model: SpMT

As our HMT work primarily targets non-numeric applications, which are the hardest to parallelize, we use

SpMT as the thread sequencing model. However, threads are formed at two different granularities. The control flow graph (CFG) is partitioned into *supertasks*, which are again partitioned into *tasks*. A task is a group of basic blocks and can have multiple targets. A supertask is a group of tasks at a macro level, which can be thought of as a bigger subgraph of the CFG. A supertask represents a substantial partition of program execution, the idea being that there is little if any control dependence between supertasks, and ideally only minimal data dependence. Thus, we have three hierarchical levels of nodes in a CFG: basic blocks, tasks, and supertasks.

Figure 3 shows a partition of a CFG into tasks and supertasks. In this paper, tasks are formed as done for the multiscalar processor in [4]. Supertasks are dynamically generated as a collection of tasks.

### 3.3. Inter-Task and Inter-Supertask Communication

This communication refers to forwarding data values across tasks and supertasks. Like many other SpMT models, the HMT model uses a *shared register space* and a shared memory address space. Inter-task communication happens implicitly due to reads and writes to the shared registers (and to shared memory locations).

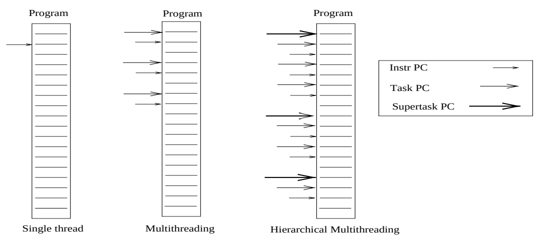

**Figure 2. Sequencing Models**

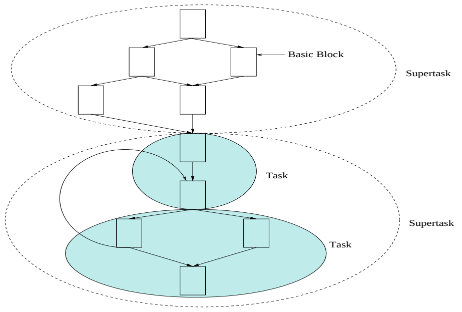

**Figure 3. Basic Blocks, Tasks, and Supertasks in a CFG**

## 4. HMT Microarchitecture

The previous section presented the concept of hierarchical multithreading. In this section, we give an overview of the microarchitecture used to study the HMT concept. This is followed in the next section by experiments and detailed analysis of the obtained results. The main purpose of this microarchitecture is for testing the HMT idea, and is by no means a final microarchitecture.

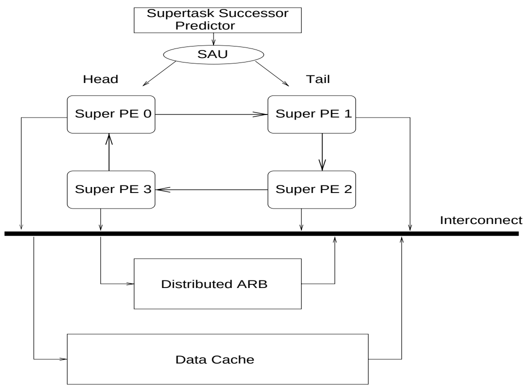

**Figure 4. Higher Level Block Diagram**

### 4.1. Higher Level

Figure 4 presents a block diagram of the higher level of the hierarchy. The superPEs are organized as a circular queue, with head and tail pointers, such that at any point of time the active superPEs are between the head and the tail. A *supertask allocation unit (SAU)* assigns supertasks to the superPEs. Initially, all the superPEs are idle, with the head and tail pointers pointing to the same superPE. The SAU assigns the first supertask to the head superPE, and advances the tail pointer to the next one in the circular queue. The *supertask successor predictor* then predicts the next supertask to be assigned. In the next cycle, the predicted successor supertask is assigned to the next superPE. This process is repeated until all of the superPEs are busy. Assigning a supertask to a superPE means submitting the starting PC of the supertask to the superPE.

Each superPE executes the supertask assigned to it. Only the head superPE is allowed to physically commit, and update the architected state. All the others buffer their values. When the head superPE commits, the head pointer is advanced to the next superPE. At that time, a check is done to see if the supertask prediction done for the new head superPE is correct or not. If the prediction turns out to be wrong, then all the supertasks from the new head until the tail are squashed, and the correct supertask is assigned to the new head.

### 4.2. Lower Level

The lower level of the HMT hierarchy is almost identical to that of a multiscalar processor, as described in [4]. The internals of a superPE are shown in Figure 5. It consists of a group of PEs, each of which can be considered to be a small superscalar processor with a small instruction window, small instruction issue, etc. These PEs are connected as a circular queue with head and tail pointers, similar to the higher level. The circular queue imposes a sequential order among the PEs, with the head pointer indicating the oldest active PE in the superPE.

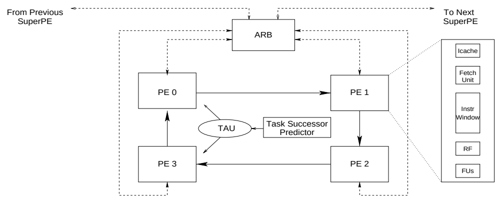

**Figure 5. Block Diagram of a SuperPE**

A distributed ARB [5] is used to enforce memory reference ordering between all PEs in all superPEs.

As the tasks within a supertask follow sequential ordering, and share a common register space, inter-PE register synchronization will happen automatically when inter-PE register communication is handled properly. In order to support fast register access, the register file is replicated as in other SpMT proposals [4][8]. With replication, a physical copy of the register file is provided in each PE. These register file replica maintain different *versions* of the register space; i.e., they store register values that correspond to the processor state at different points in a sequential execution of the program.

## 5. Experimental Evaluation

Next, we perform a detailed quantitative evaluation of the HMT concept. It is important to note that we are not assessing the HMT microarchitecture given in the previous section, but the HMT concept. Thus there may be better microarchitectures for this concept, but the one presented here will provide significant results upon which we can base our conclusions about the feasibility and performance of the hierarchical model, and hence the potential to build better microarchitectures.

### 5.1. Experimental Methodology and Setup

Our experimental setup consists of a detailed cycle-accurate execution-driven simulator based on the MIPS-I ISA. The simulator accepts executable images of programs, and does cycle-by-cycle simulation; it is not trace driven. The simulator faithfully models the above architecture; all important features of the HMT processor, including the superPEs, the PEs within the superPEs, execution along mispredicted paths, inter-PE & inter-superPE register communication, and inter-PE & inter-superPE memory communication have been included in the simulator. The simulator is parameterized; we can vary the number of superPEs, the number of PEs in a superPE, the PE issue width, the task size, and the cache configurations. Some of the hardware parameters are fixed at the default values given in Table 1. The parameters on the left hand side of the table are specific to a PE, and those on the right are for the entire processor. The successor predictor we use is similar to a two-level data value predictor [14].

For benchmarks, we use a collection of 7 programs, five of which are from the SPECint95 suite. The programs are compiled for a MIPS R3000-Ultrix platform with a MIPS C (Version 3.0) compiler using the optimization flags distributed in the SPEC benchmark makefiles. We ran our simulations up to 100 millions instructions.

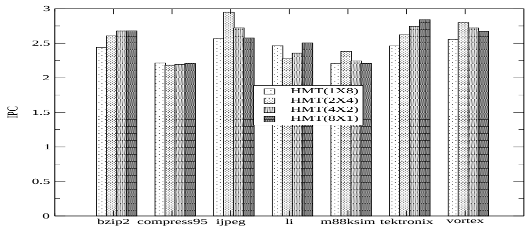

**Figure 6. IPC for Different Configurations**

**Simulated Configurations:** For denoting the HMT configurations, we use the notation HMT($m \times n$) for a configuration of $m$ superPEs each of which has $n$ PEs. We are simulating HMT(1×8),HMT(2×4),HMT(4×2) and HMT(8×1). The main difference between HMT(1×8) and HMT(8×1) is that in the former, after the assigned supertask physically commits, another one is assigned, while in the latter there is no notion of supertasks. Thus HMT(8×1) does not suffer from the overhead of waiting till the assigned supertask commits before assigning another one.

**Task and Supertask Formation:** Tasks are formed at compile time by going through the static binary sequentially and adding instructions to the task until 32 instructions are reached, or a call/return instruction is reached, or the number of targets becomes 4. The task information is conveyed to the hardware. For this paper, supertasks are formed as a group of X tasks, where X is the number of PEs in a superPE. This happens as follows: A PC is assigned to the local sequencer of a superPE. The task predictor and the thread allocation unit (TAU) of the superPE assign tasks to the PEs of the superPE until no PEs are idle within this superPE. The starting PC of the head task, along with the different targets, are stored in a specific table with a unique ID for the supertask. An extensive comparison of different supertask formation strategies is beyond the scope of this paper.

### 5.2. Experimental Results and Analysis

In this section we present experimental results we have obtained. Figure 6 shows the IPC (Instructions Per Cycle) obtained for the above configurations. For `bzip2`, HMT(4×2) and HMT(8×1) have roughly the same performance. As for `compress95`, HMT(1×8), HMT(4×2), and HMT(8×1) are very close. HMT(2×4) has much better IPC for `ijpeg`, `m88ksim`, and `vortex`. HMT(8×1) is better for `li` and `tektronix`. To understand these results we must analyze them based on the following parameters: (1)task prediction, (2)supertask prediction and (3)register values across superPEs.

Table 2 shows the task prediction accuracy, and Table 3 shows the supertask prediction accuracy for the last three

| **Default Values for Simulator Parameters** | | | |
|---|---|---|---|
| *PE Parameter* | Value | *Processor Parameter* | Value |
| *Max task size* | 32 instructions | | |
| *PE issue width* | 2 instructions/cycle | | |
| *Task predictor* | 2-level predictor 1K entry, pattern size 6 | *Supertask predictor* | 2-level predictor 1K entry, pattern size 6 |
| *L1 - Icache* | 16KB, 4-way set assoc., 1 cycle access latency | *L1 - Dcache* | 128KB, 4-way set assoc., 2 cycle access latency |
| *Functional unit latencies* | Integer/Branch :- 1 cycle Mult/Divide :- 10 cycles | | |

**Table 1. Default Parameters for the Experimental Evaluation**

| Benchmark | Task Prediction Accuracy for | | |
|---|---|---|---|
| | HMT(1×8) | HMT(2×4) | HMT(4×2) |
| bzip2 | 99.43% | 99.06% | 98.64% |
| compress95 | 89.54% | 80.91% | 79.06% |
| ijpeg | 96.24% | 93.43% | 93.45% |
| li | 89.19% | 79.89% | 75.99% |
| m88ksim | 99.89% | 99.88% | 99.87% |
| tektronix | 94.72% | 91.78% | 90.17% |
| vortex | 97.52% | 96.47% | 95.12% |

**Table 2. Task Prediction Accuracy**

| Benchmark | Supertask Prediction Accuracy for | | |
|---|---|---|---|
| | HMT(2×4) | HMT(4×2) | HMT(8×1) |
| bzip2 | 98.18% | 98.36% | 96.38% |
| compress95 | 60.65% | 84.85% | 93.43% |
| ijpeg | 87.01% | 89.66% | 92.51% |
| li | 57.33% | 82.21% | 92.61% |
| m88ksim | 99.48% | 90.30% | 95.39% |
| tektronix | 98.05% | 90.33% | 96.00% |
| vortex | 79.52% | 87.98% | 94.83% |

**Table 3. Supertask Prediction Accuracy**

| Benchmark | Config. | No. of Distinct Supertasks | Successor Supertasks | |
|---|---|---|---|---|
| | | | Average Number | Highest Number |
| bzip2 | HMT(2×4) | 120 | 1.70 | 7 |
| | HMT(4×2) | 151 | 1.42 | 6 |
| | HMT(8×1) | 209 | 1.21 | 7 |
| compress95 | HMT(2×4) | 158 | 2.39 | 14 |
| | HMT(4×2) | 210 | 1.60 | 7 |
| | HMT(8×1) | 256 | 1.34 | 8 |
| ijpeg | HMT(2×4) | 349 | 1.80 | 11 |
| | HMT(4×2) | 435 | 1.48 | 15 |
| | HMT(8×1) | 611 | 1.27 | 18 |
| li | HMT(2×4) | 600 | 2.95 | 38 |
| | HMT(4×2) | 680 | 2.08 | 52 |
| | HMT(8×1) | 788 | 1.53 | 54 |
| m88ksim | HMT(2×4) | 199 | 1.95 | 10 |
| | HMT(4×2) | 248 | 1.59 | 8 |
| | HMT(8×1) | 333 | 1.31 | 4 |
| tektronix | HMT(2×4) | 479 | 2.13 | 24 |
| | HMT(4×2) | 572 | 1.62 | 27 |
| | HMT(8×1) | 681 | 1.34 | 24 |
| vortex | HMT(2×4) | 3022 | 2.42 | 82 |
| | HMT(4×2) | 3496 | 1.84 | 139 |
| | HMT(8×1) | 4096 | 1.50 | 188 |

**Table 4. Supertask Statistics**

configurations (note that HMT(1×8) does not use supertask prediction, HMT(8×1) does not use task prediction). We can deduce the following.

It is shown that from the two non-hierarchical configurations, HMT(1×8) performs worse, because of the overhead introduced by delaying supertask assignment until the previously assigned supertask has been committed.

`compress95` and `li` have low supertask prediction accuracy, therefore the hierarchical model does not perform well for them.

For `ijpeg`, `m88ksim` and `vortex`, HMT(2×4) performs the best, because it has the highest supertask prediction accuracy (although it does not have the highest task prediction accuracy). This means that for the hierarchical model, supertask prediction accuracy is more crucial than task prediction accuracy. This is because task misprediction only causes the squashing of PEs inside the superPE that did the misprediction, whereas supertask misprediction leads to the squashing of all subsequent superPEs.

Another factor that affects the supertask prediction accuracy and hence the overall performance is the distribution of supertasks and their successors; that is, the average number of successors per supertask and the number of distinct supertasks executed. This is because if the average number of successors is high, or if there are some supertasks with large number of successors, then the supertask predictor is likely to perform poorly. Currently the task size is 32 instructions and the supertask size is `X` tasks where `X` is the number of PEs per superPE. A more efficient scheme is needed to get better performance; that is, we need a scheme that forms supertasks with a small number of successors.

One factor that affects HMT's performance is register forwarding across superPEs. The results presented here are based on the following strategy. In a clock cycle, if any PE in a superPE modified a register, this new value is forwarded to the successor superPE. Since it is very likely that a register will be modified several times in a supertask, a lot of re-execution will occur in the receiving superPE, which lowers the performance. Thus a better mechanism is needed, for example forwarding a value when it is known that it is the last update within the superPE. Another option is to use data value prediction to predict all incoming register values for a supertask when the supertask is assigned to a superPE. Later, actual register values can be forwarded when they become available. We have conducted some preliminary experiments using data value prediction (with a hybrid(stride, context) predictor) for some benchmarks and have obtained encouraging results. For example, `bzip2`'s IPC for HMT(2×4) increased from 2.61 to 2.85, and for HMT(4×2) increased from 2.68 to 3.19. These results are not reported in tabular form due to lack of space.

Table 4 shows some statistics about supertasks for HMT(2×4), HMT(4×2) and HMT(8×1). As the number of PEs per superPE decreases, the granularity of our supertasks decreases, and the number of distinct supertasks executed increases. Also, as we increase the size of the supertask (which is currently the number of PEs in a superPE), the average number of distinct successors increases, which means that the strategy we have used for forming the supertask is not good, and needs to be enhanced, maybe with some help from the compiler. Another point worth noting here is that the average number of successors is not the only factor to consider, but also the distribution of these successors and the frequency of occurrence of each supertask. For example, if a supertask has 2 successors each of which occurs 32 times, and another supertask has 2 successors one of which occurs 2 times and the other 62 times, then in the second case the prediction accuracy will be better. All of these factors must be taken into account while forming the supertasks in order to get the best performance.

We conducted many more studies, the results of which are not presented here for lack of space. These studies confirm that whenever the hierarchical scheme performs worse than the non-hierarchical one, the main cause is bad supertask distribution. That is: (1) The frequencies of occurrence of distinct supertasks are very close. (2) The frequencies of occurrence of a supertask's successors are very close (as indicated in the above example).

## 6. Conclusion

In this paper we have investigated the feasibility of hierarchical multithreading (HMT) and the conditions for getting best results from this scheme. We summarize our observations below:

HMT is a good candidate scheme for exploiting parallelism at different granularities.

Having good supertask prediction accuracy is crucial for HMT, otherwise a non-hierarchical model is better. Thus if there is an architecture that senses the supertask prediction accuracy and adjusts its configuration from hierarchical to non-hierarchical in a dynamic manner, it can provide good performance. Supertask formation is another crucial factor for HMT, as it affects the successor prediction accuracy and hence the overall performance.

Efficient register forwarding mechanism is needed in order to avoid a lot of re-execution. Data value prediction is a good candidate for enhancing the overall performance.

## Acknowledgements

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

## References

[1] H. Akkary and M. A. Driscoll, "A Dynamic Multithreading Processor," in Proc. 31st Int'l Symposium on Microarchitecture (MICRO-31), 1998.

[2] D. Culler, J. P. Singh, and A. Gupta, "Parallel Computer Architecture: A Hardware/Software Approach," Morgan Kaufmann Publishers, 1998.

[3] P. Faraboschi, G. Desoli, and J. Fischer, "Clustered Instruction-level Parallel Processors," Tech. Rep., HP Laboratories, 1998.

[4] M. Franklin, The Multiscalar Processor. PhD thesis, University of Wisconsin-Madison, 1993.

[5] M. Franklin and G. S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," IEEE Transactions on Computers, vol. 45, no. 5, pp. 552–571, 1996.

[6] V. Krishnan and J. Torrellas, "Executing sequential binaries on a clustered multithreaded architecture with speculation support," in Proc. Int'l Conf. on High Performance Computer Architecture (HPCA), 1998.

[7] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," in Proc. 19th Int'l Symposium on Computer Architecture, pp. 46–57, 1992.

[8] P. Marcuello and A. Gonzalez, "Clustered Speculative Multithreaded Processors," in Proc. Int'l Conf. on Supercomputing, pp. 20–25, 1999.

[9] I. Martel, D. Ortega, E. Ayguade, and M. Valero, "Increasing Effective IPC by exploiting Distant Parallelism," in Proc. Int'l Conf. on Supercomputing, pp. 348–355, 1999.

[10] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, "Trace processors," in Proc. 30th Annual Symposium on Microarchitecture (MICRO-30), pp. 24–34, 1997.

[11] G. S. Sohi, S. Breach, and T. N. Vijaykumar, "Multiscalar Processors," in Proc. 22nd Int'l Symposium of Computer Architecture, pp. 414–425, 1995.

[12] J.-Y. Tsai et.al, "Integrating Parallelizing Compilation Technology and Processor Architecture for Cost-Effective Concurrent Multithreading," Journal of Information Science and Engineering, vol. 14, March 1998.

[13] D. W. Wall, "Limits of Instruction Level Parallelism," Tech. Rep., Western Research Laboratory, 1993.

[14] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," in Proc. 30th Int'l Symposium on Microarchitecture (MICRO-30), pp. 281–290, 1997.