Hierarchical Multi-Threading For Exploiting Parallelism at Multiple Granularities

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

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

Abstract

As we approach billion-transistor processor chips, the need for a new architecture to make efficient use of the increased transistor budget arises. Many studies have shown that significant amounts of parallelism exist at different granularities that is yet to be exploited. Architectures such as superscalar and VLIW use centralized resources, which prohibit scalability and hence the ability to make use of the advances in semiconductor technology. Decentralized architectures make a step towards scalability, but are not geared to exploit parallelism at different granularities. In this paper we present a new execution model and microarchitecture for exploiting both adjacent and distant parallelism in the dynamic instruction stream. Our model, called hierarchical multi-threading, uses a two-level hierarchical arrangement of processing elements. The lower level of the hierarchy exploits instruction-level parallelism and thread-level parallelism, whereas the upper level exploits more distant parallelism. Detailed simulation studies with a cycle-accurate simulator indicate that the hierarchical multithreading model provides scalable performance for most of the non-numeric benchmarks considered.

Keywords: Speculative multithreading, Control independence, Microarchitecture, Thread-level Parallelism, parallelism granularity

1 Introduction

A defining challenge for research in computer science and engineering has been the ongoing quest for faster execution of programs. The commodity microprocessor industry has been traditionally looking to fine-grain or instruction level parallelism (ILP) for improving performance, by using sophisticated microarchitectural techniques and compiler optimizations. These techniques have been quite successful in exploiting ILP.

Many proposals such as the multiscalar [3][13], trace processing [8], superthreading [14], and clustered multithreading [2][6] have been proposed to reduce the cycle time and to exploit thread level parallelism. All of these proposals are geared to exploiting thread-level parallelism at one granularity. In this paper we investigate a hierarchical multithreading model to exploit thread-level parallelism at two granularities. It makes use of decentralization and multithreading to extract both fine- and medium-grain parallelism (also known as ILP and TLP). This execution model has the potential for better scalability of performance than non-hierarchical multithreading execution models.

The rest of the paper is organized as follows. Section 2 presents the background and related work. Section 3 describes the HMT (Hierarchical Multi-Threading) thread model. Section 4 presents a detailed description of the HMT microarchitecture. Section 5 presents experimental results obtained from detailed simulations of a cycle-accurate HMT simulator. Finally we conclude in section 6, followed by a list of references.

2 Background and Related Work

Limited size instruction window is one of the major hurdles in exploiting parallelism. Programs are hundreds of millions of instructions. 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 increases the visibility of the program structure at run time. However, having a very large instruction window is difficult, for the following reasons: (i) Implementation constraints limit the window size. (ii) Branch misprediction reduces the number of useful instructions in the instruction window. (iii) Having a large instruction window increases the complexity of the instruction scheduler, thus increasing the cycle time.

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 *parallel threads* model, threads that execute in parallel are control-independent, and the decision to execute a thread does not depend on the other active threads. 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, which envisions a strict sequential ordering among the threads. That is, 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. This relaxation makes it possible to "parallelize" non-numeric applications into threads without explicit synchronization, even if there is a potential inter-thread data dependence. 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.

Examples of prior proposals using sequential threads are the multiscalar model [3][13], the superthreading model [14], the trace processing model [8], and the dynamic multithreading model [1]. In the sequential threads model, threads can be *nonspeculative* or *speculative* from the control point of view. 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 [3][5][6][8][13][14].

The speculative multithreading architectures discussed so far use a single level of multi-threading. The program is partitioned into a set of 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. 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. Thus, if it exploits fine-grain TLP, then it does not exploit more distant parallelism, and vice versa. In order to obtain high performance, we need to extract parallelism at different granularities.

A second drawback of single-level multithreading is that it is difficult to exploit control independence between multiple threads. If there is a thread-level misprediction, then all subsequent threads beginning from the mis-speculated thread are generally squashed, even though some threads may be control independent on the misspeculated one. It is possible to modify the hardware associated with the circular queue in order to take advantage of control independence [9], however the design becomes more complicated.

3 The HMT Thread Model

We investigate *hierarchical multithreading (HMT)* to overcome the limitations of single-level multithreading. This section introduces the software aspects of our HMT model; the next section details the microarchitecture aspects. An important attribute of any multithreading system is its thread model, which specifies the sequencing of threads, the name spaces (such as registers and memory addresses) threads can access, and the ordering semantics among these operations, particularly those done by distinct threads.

As our HMT work primarily targets non-numeric applications, we use SpMT as its thread sequencing model. However, threads are formed at two different granularities. The control flow graph 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 control flow graph. 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. Generally, instructions in two adjacent supertasks are far away in the dynamic instruction stream and have a high probability of being mutually control independent. Thus, we have three hierarchical levels of nodes in a CFG: basic blocks, tasks, and supertasks.

The criterion used for this partitioning is important, because an improper partitioning could in fact result in high inter-thread communication and synchronization, thereby degrading performance! True multithreading should not only aim to distribute instructions evenly among the threads, but also aim to minimize inter-thread communication by localizing a major share of the inter-instruction communication occurring in the processor to within each PE. In order to achieve this, mutually data dependent instructions are most likely allocated to the same thread.

In this paper, tasks are formed as done for the multiscalar processor in [3]. Supertasks are dynamically generated as a collection of tasks. This is done as follows: the task predictor begins assigning tasks to PEs of a superPE. As soon as all the PEs of the superPE are running these tasks are assumed to be a supertask,

given a unique ID and stored in a specific table. For the time being, each supertask is composed of fixed number of tasks. Thus, is task prediction is used in order to generate the required number of tasks per supertask. This may not be the best strategy but is used for the time being in order to test our architecture.

4 The HMT Microarchitecture

In this section we describe one possible microarchitecture for our HMT thread model. In order to parallelly execute multiple tasks and supertasks in an efficient manner, we investigate a two-level hierarchical multithreaded microarchitecture. The higher level is composed of several *superPEs*, and is used for executing multiple supertasks in parallel. At the lower level of the hierarchy, each superPE consists of several PEs, each of which executes a task. Figure 1 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 global sequencer assigns supertasks to the superPEs.

4.1 Program Execution in the HMT Processor

Initially, all the superPEs are idle, with the head and tail pointers pointing to the same superPE. The global sequencer 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 successor of the supertask just assigned. (Our experiments show that control flow between supertasks is highly repetitive.) 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. Currently we set the size of a supertask to be equal to X tasks where X is the number of PEs per 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, as will be shown in detail later. 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.

The above description is for the higher level of the hierarchy, the one involving supertasks and superPEs.

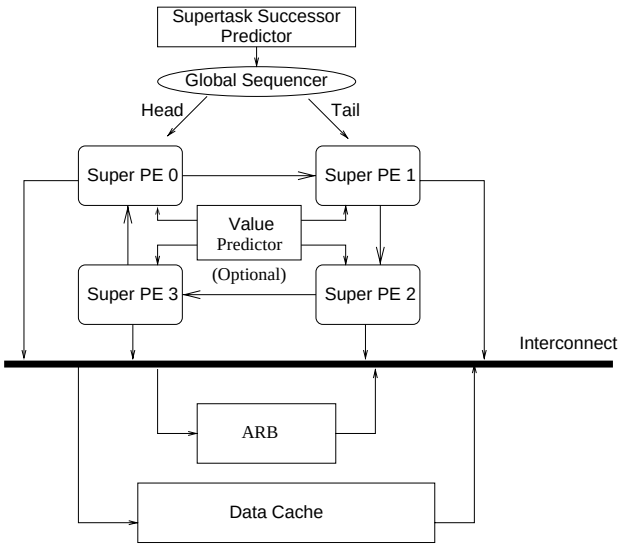

Figure 1: The HMT Processor

Next, we shall see how each supertask is executed within a superPE.

4.2 Lower Level of HMT Microarchitecture: SuperPE

The lower level of the HMT hierarchy considered in this paper is almost identical to a multiscalar processor, as described in [3]. The internals of this level are (briefly) described for the benefit of readers who are unfamiliar with the details of the multiscalar processor. Due to space limitations, this description is kept brief; interested readers are encouraged to consult [3] for the details. The internals of a superPE are shown in Figure 2. It consists of a group of PEs, each of which can be considered 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.

A local sequencer with a local task successor predictor is responsible for assigning tasks to the PEs. When the tail PE is idle, a sequencer invokes the next task (as per sequential ordering) on the tail PE, and advances the tail pointer. When a task prediction is found to be incorrect, all subsequent tasks within the superPE are squashed; supertasks executing in subsequent superPEs are not squashed. Completed tasks are retired from the head of the PE queue, enforcing the required sequential ordering within the supertask. This retirement is *speculative*, if the superPE is not the current head of the higher level of the hierarchy.

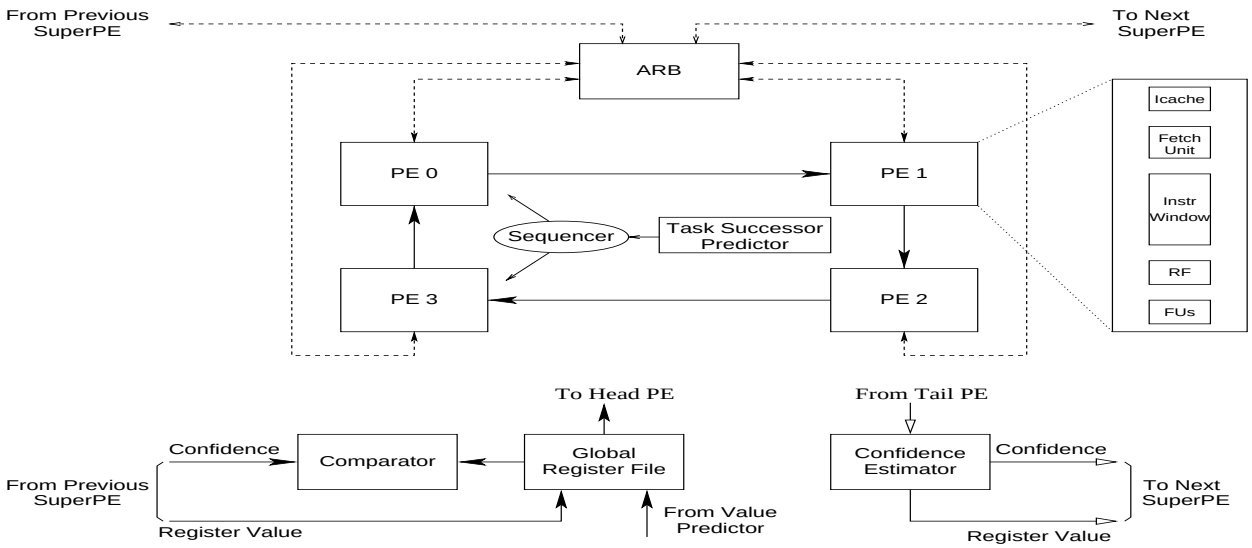

Figure 2: Block Diagram of a SuperPE

4.3 Inter-superPE Register Communication

Next, let us consider communication of register values from one supertask to another, at the higher level of the HMT hierarchy. As shown in Figure 2, each superPE maintains a *global register file* to store the supertask's incoming register values (from the previous superPE or the data value predictor). When a new register value arrives at a superPE, this value is compared against the existing value for that register in the global register file. If there is a change, then the new value is forwarded to its head PE, from where it gets forwarded to subsequent PEs, if required.

A superPE receives two sets of register values, one coming from the previous superPE (in case the current one is not the head superPE) and the other from the data value predictor. The choice between these values is done based on the confidence values. Each register value, whether coming from the value predictor or the predecessor superPE, has its own confidence value, and the superPE chooses the value with the highest confidence. Each superPE also has a confidence estimator to calculate the confidence values for the register values sent to its successor, as shown in Figure 2. Of course, if the predecessor superPE is the head and is committing, its values have the highest confidence.

The confidence estimations coming from the value predictor, are derived from the saturating counters of the predictor. The confidence of the values coming from the predecessor depends on the distance of the predecessor from the head superPE, the number of cycles since the assignment of the supertask to the predecessor as well as whether registers have been modified by the predecessor. In order to optimize inter-superPE register communication, we need to address two questions: (1) how often are register values sent to a superPE (whether from the data value predictor or from the predecessor superPE)? (2) which values should be sent?

For the first question, the following scheme is applied: the data value predictor predicts values for all registers only once for each supertask, and this is done at the time the supertask is assigned to a superPE. Each time a superPE wants to send values to its successor, it first calculates confidence for its values and then sends them. The receiving superPE will compare the confidence values of the new register values against the existing confidence values. If the new confidence value is higher for a particular register, then the new value is accepted. Register values are passed from a superPE to its successor when the successor superPE is about to speculatively commit its first task, which is not too early so the superPE does not use obselete values nor too late so the superPE would not have done a lot of useless work.

Next we address the second question of which value exactly to send. Sending all the register values has two drawbacks: (1) high bandwidth requirement, (2) low utility, as all registers may not have new values. Therefore, all registers are communicated only the first time. As time progresses since a supertask started execution, register values generated by that supertask tend to have higher confidence values and only modified registers are communicated to the subsequent superPEs. Also, all register values are communicated from the

head superPE when it is about to physically commit.

4.3.1 Data Value Prediction

The HMT microarchitecture can benefit from data value prediction, which involves predicting the incoming register values for a supertask. One option for the data value predictor is a hybrid predictor that uses 2-delta stride predictor [11] as well as context based predictor [12]. This hybrid predictor can be modified to predict all register values at once, in a way similar to [10].

Without the data value predictor, the only source of information available will be the premature register values coming from the predecessor superPE. Because these will most likely change during execution, the newly assigned supertask is likely to do a lot of useless work, if data value prediction is not used.

The job of the data value predictor is: given a supertask ID, predict *all* register values at the same time, together with confidence values for each predicted value. The predictor is updated each time a supertask is physically committed by the head superPE.

The data value predictor has two tables: SHT (Supertask History Table) and VPT (Value Prediction Table). Each SHT entry contains the following fields: (i)Frequency of accessing the entry, in order to use *least frequently used* replacement policy. (ii)Tag field, (iii)For each architected register it has: Last k values produced for this register, confidence estimator for the stride and predictor type (stride or context) that made the last prediction.

The last two values in the history of a register are used to make a prediction using a delta stride predictor. The confidence estimator of the delta stride predictor is simply a saturating counter that is incremented in case of correct prediction and decremented otherwise. The last k values are combined using a hash function and are used to index the VPT that contains k saturating counters for those values. The value with the highest counter is picked as the prediction of the context based predictor, with the corresponding counter as the confidence estimator. From the values predicted by the stride and context components, we pick the one with higher confidence. In the case of a misprediction, the counters corresponding to the predictor that made the last prediction are decremented. Similarly, in the case of a correct prediction, the corresponding counters are incremented. If the confidence estimators happen to be the same, one of the two values is selected at random.

4.4 Inter-SuperPE Memory Communication

At the higher level of the HMT hierarchy, inter-superPE memory communication is done by connecting the Address Resultion Buffers (ARBs) [4] of each superPE by a bidirectional ring. Thus, the memory data dependence speculation part is distributed at the higher level. When a load reference is issued in a superPE, and its ARB does not contain a prior store to the same location, the request is forwarded to the predecessor superPE's ARB, and so on. Similarly, when a store is issued in a superPE, it will be forwarded to the successor superPE's ARB, if no subsequent stores have already been issued to the same address from its superPE.

4.5 Advantages of the HMT Paradigm

Task mispredictions typically cause squashing only within its superPE. Data dependence violations only cause re-execution of the affected instructions.

The benefits of HMT stem from two important features: program characteristics and technological aspects. Program characteristics reveal the following: (i) Studies have shown that parallelism is there [7][15], but most of it cannot be exploited due to the large distance in the dynamic instruction stream. Our proposed architecture exploits some of the distant parallelism by executing supertasks in parallel. (ii) Multiscalar studies show IPC (Instructions Per Cycle) to be tapering off as more and more PEs are added [13]. This is because, in the case of a task misprediction, all the PEs starting from the PE with the misprediction will be squashed, thus decreasing the percentage of PEs doing useful work. We try to avoid this by letting each superPE have a small number of PEs and assign control-independent supertasks to multiple superPEs, as much as possible.

Also, if we squash a PE in a superPE, only the subsequent PEs in the same superPE are squashed; the remaining PEs in the other superPEs are not squashed (unless there is a change in successor supertask).

An important point to note is that the hierarchical arrangement of PEs as in the HMT microarchitecture does not require substantial additions or complexity to the hardware. The main newly introduced hardware is the confidence estimators for the register values and the comparators for comparing them. The data value predictor and the supertask predictor are two other newly introduced hardware structures. Apart from these, there is little new hardware. In fact, the two-level hierarchy can be dynamically reconfigured as a flat multithreaded processor, if required.

Default Values for Simulator Parameters			
PE		*Processor*	
Parameter	*Value*	*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	Int/Branch :- 1 cycle Mul/Div :- 10 cycles	*Data value predictor*	hybrid(stride, context), k=4 SHT 1K entries and VPT 64K entries

Table 1: Default Parameters for the Experimental Evaluation

5 Experimental Evaluation

The previous section presented a detailed description of an HMT microarchitecture. Next, we present a detailed quantitative evaluation of this processing model. Such an evaluation is important to study its performance characteristics, and to see how scalable this architecture is.

5.1 Experimental Methodology and Setup

Our experimental setup consists of a detailed cycle-accurate execution-driven simulator based on the MIPS-II ISA. The simulator accepts executable images of programs, and does cycle-by-cycle simulation; it is not trace driven. The simulator faithfully models the HMT architecture depicted in Figures 1 and 2; 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 [16].

For benchmarks, we use a collection of 7 programs, five of which are from the SPEC95 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. The benchmarks are simulated up to 100 million instructions each.

Our simulation experiments measure the execution time in terms of the number of cycles required to execute a fixed number of instructions. While reporting the results, the execution time is expressed in terms of instructions per cycle (IPC). The IPC values include only the committed instructions, and do not include the squashed instructions.

5.2 Experimental Results

Our first set of experiments are intended to show the benefits of the hierarchical arrangement. For this purpose, we simulate an HMT(3×4) processor (that is, an HMT processor with 3 superPEs, each of which has 4 PEs) as well as an HMT (1×12) processor, *both of which do not use data value prediction*, in order to show the potential of the HMT without the advantage of data value prediction. These results are presented in Figure 3. On the X-axis, we plot the benchmarks, and on the Y-axis, we plot the IPC values. Each benchmarks has two histogram bars, corresponding to HMT(3×4) and HMT (1×12), respectively.

The first thing to notice in Figure 3 is that the HMT(3×4) architecture is performing better than the non-hierarchical HMT(1×12) architecture for 5 of the 7 benchmarks. Looking at specific benchmarks, the HMT architecture performs relatively the best for `bzip2` and `ijpeg`. It performs relatively worse for `compress95` and `li`. Among these two, `compress95` does not have much parallelism. `li` has notable amounts of parallelism; on analyzing the results for `li`, we found that the prediction accuracy for procedure returns was low. We intend to investigate better predictors for predicting the return addresses in a hierarchical setting.

We next present some run-time statistics for the HMT(3×4) configuration; these statistics are somewhat different for the different configurations. Table

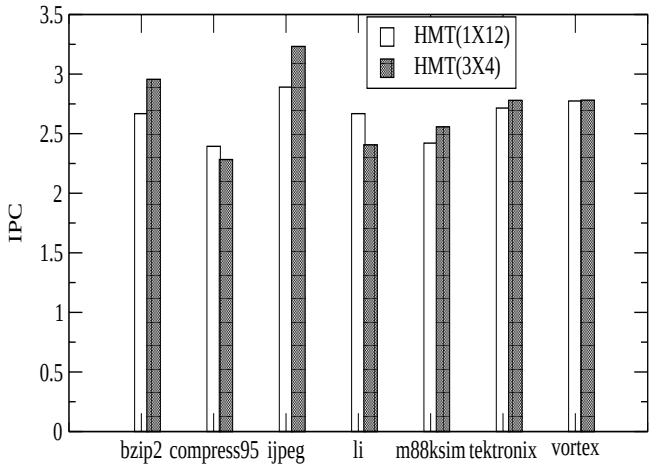

Figure 3: Performance of an HMT(3 × 4) Processor and an HMT(1 × 12) Processor, without Data Value Prediction

2 presents these statistics. In particular, it presents the data value prediction accuracy, supertask prediction accuracy, average active superPEs in a cycle, and the number of distinct supertasks executed.

Next, we perform sensitivity studies to study the performance of the HMT architecture under various conditions. In particular, we experiment with two different values for the number of PEs per superPE — 2 and 3. For each of these values, we vary the number of superPEs from 1 to 3. These studies use data value prediction at the higher level; performance does not scale very well when data value prediction is not used. Figure 4 presents the results of these sensitivity studies. The left graph is for 2 PEs/SuperPE and the right one is for 3 PEs/SuperPE.

From the results presented in Figure 4, we can see that except for `compress95` and `li`, all of the other benchmarks show good scalability in performance when the number of superPEs is increased from 1 to 3. That is, even when we use a total of 12 PEs, we still get reasonably good performance.

6 Summary and Conclusions

This paper presents a two-level hierarchical architecture that exploits parallelism at different granularities. The processing elements are organized in ring of rings, rather than a single ring. While each smaller ring (*superPE*) executes a sequence of tasks (one per PE) as in the Multiscalar, a high-level sequencer assigns *supertasks* to the superPEs. Such an architecture addresses several key issues, including maintaining a large instruction window, while avoiding centralized resources and minimizing wire delay.

Detailed simulation results show that for most of the non-numeric benchmarks used, the hierarchical approach provides better performance than the non-hierarchical approach. The results also show that a small percentage of the programs may not benefit from a hierarchical multithreading execution model.

The HMT microarchitecture presented in this paper is just one way to exploit parallelism at multiple granularities. Future work involves integrating compiler support. We intend to start with a post-compilation step to generate supertasks that are roughly of the same size, are somewhat control independent, and are somewhat data independent. We also intend to explore the memory hierarchy and memory disambiguation system to find the best model for the HMT model.

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*, 1998.

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

[3] M. Franklin, *The Multiscalar Architecture*. PhD thesis, Technical Report 1196, Computer Science Department, University of Wisconsin-Madison, 1993.

[4] 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.

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

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

[7] 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.

[8] 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.

Benchmark	Data Value Pred. Accuracy	Supertask Pred. Accuracy	Avg. Active SuperPEs per cycle	# Distinct Supertasks Executed
bzip2	83.4%	94.7%	2.88	119
compress95	46.3%	88.0%	2.64	164
ijpeg	69.9%	83.9%	2.67	348
li	65.0%	87.7%	2.06	628
m88ksim	91.6%	96.7%	2.83	209
tektronix	68.7%	89.3%	2.55	494
vortex	67.6%	74.7%	2.79	2994

Table 2: Run-Time Statistics for HMT(3 × 4) Configuration

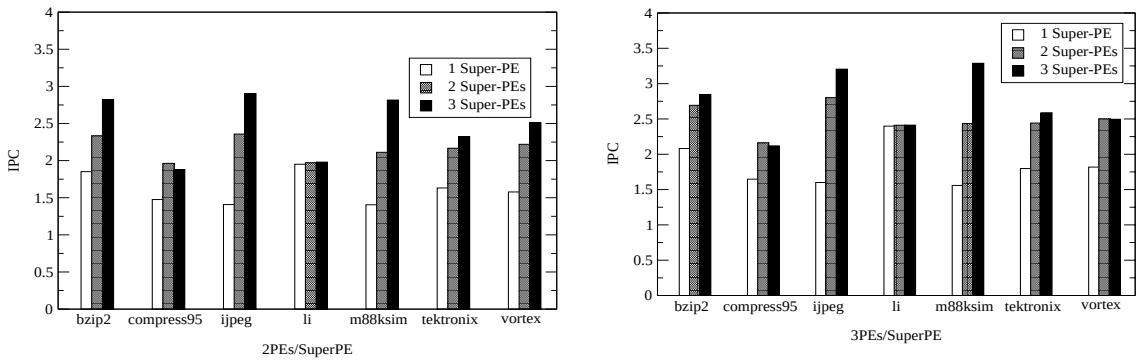

Figure 4: Performance of HMT Architecture for Different Configurations

[9] E. Rotenberg, and J. E. Smith, "Control Independence in Trace processors," in Proc. *32nd Int'l Symposium on Microarchitecture (Micro-32)*, 1999.

[10] R. Sathe, K. Wang, and M. Franklin, "Techniques for performing highly accurate data value prediction," *Microprocessors and Microsystems*, 1998.

[11] Y. Sazeides and J. E. Smith, "The Predictability of Data Values," in *Proc.30th Int'l Symposium on Microarchitecture*, pp. 248–258, 1997.

[12] Y. Sazeides and J. E. Smith, "Implementations of Context based Value Predictors," Tech. Rep., University of Wisconsin-Madison, 1997.

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

[14] J.-Y. Tsai and et.al, "Integrating parallelizing compilation technology and processor architecture for cost-effective concurrent multithreading," *Journal of Information Science and Engineering*, vol. 14, March 1998.

[15] S. Vajapeyam, P. J. Joseph, and T. Mitra, "Dynamic Vectorization: A Mechanism for Exploiting Far-flung ILP in Ordinary Programs," in *Proc. 26th Int'l Symposium on Computer Architecture*, pp. 12–27, 1999.

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