A Survey on Helper Threads and Their Implementations

Ahren Studer

This survey covers the general idea behind helper threads and the major ways in which they are implemented. The first section covers parallel helper threads and the issues crucial to their role in improving performance. The following sections cover the implementations of parallel helper threads, focusing on SMT processor based implementations and adapted superscalar processor based implementation with the advantages and disadvantages each method provides. Last, the idea of a sequential helper thread is introduced as a context that executes while the main thread has limited ILP. In the remaining section of the paper, sequential helper thread properties and the details of its implementation are covered.

1. Introduction:

Current microprocessors provide substantial resources to exploit instruction level parallelism (ILP) and high instruction throughput. However, data and control dependences cause a much lower than ideal instructions per cycle (IPC). With control dependences, branch mispredictions cause substantial slow down to programs where the hardware predictor has difficulty predicting critical branches. To make the situation worse with the deeper pipelines used to increase the frequency of the processor and allow for more ILP, the branch misprediction penalty becomes larger as more stages of the pipeline need flushed and refilled. This higher frequency also hurts the IPC by increasing the processor-memory gap, leading to relatively longer cache miss penalties, degrading performance when large amounts of data dependences exist.

To reduce the impact of these critical instructions, loads that miss in the cache and difficult to predict branches, it is critical to note that the main program does calculate the values crucial for the address of the load or for the branch outcome, but not in a timely fashion. One attempt to improve performance with regards to memory latency is prefetching of difficult loads. However, this increases the number of instructions executed by the program and causes significant overhead if the data was in the cache. Several different heuristics have been suggested to handle the problem of difficult to predict branches. However, the balance of speed, accuracy, and space required leads to less than optimal performance. To address these problems researches have suggested the concept of a helper thread. In general, a helper thread begins execution prior to a problem instruction in the main thread, performs some type of work to lessen the penalty caused by the problem instruction without changing the state of the main thread of execution, and stops execution once the instruction has been handled by the helper thread or executed by the main thread.

Critical to a helper thread's effectiveness is the amount of overhead it incurs for the main thread of execution and whether or not the helper thread is able to calculate the value needed to resolve a critical instruction in time. Timeliness is critical since a branch prediction is useless after the branch has been resolved and a prefetch would simply be extra computation if the main thread already executed the load. To address these problems, helper threads are implemented in a way to interfere as little as possible with

the main thread while delivering results in a timely manner. To minimize interference the majority of helper threads are executed using spare contexts on an SMT processor or by altering a superscalar processor to accept instructions from both the helper thread and the main thread simultaneously. By using an SMT the complexity of implementing the helper threads is greatly reduced, but the overhead associated with communicating values to the helper thread is higher. When a superscalar processor is altered for helper threads, less overhead is involved with creation and execution, but the cost of such changes may be too great depending on which programs the system runs. One major deviation from the paradigm involves a helper thread is not run simultaneously with the main thread, but executes only when limited ILP exists in the main thread.

The remainder of this paper will focus on
Section 2: general parallel helper thread issues
Section 3: parallel helper thread implementations
Section 4: sequential helper threads
Section 5: conclusion

2. Parallel Helper Thread Issues:

2.1 Instructions Included in the Helper Thread:

The issue of which instructions are included in the helper thread is strongly associated with the manner used to create the helper thread. Some implementations save complexity and space by simply running the same code as the main thread, but add complexity by adding hardware to stop the helper thread once its work is done. Other helper threads are constructed by hand or through the use of additional hardware or software, resulting in sequences of less than thirty instructions [Collins, Wang, Tullsen, Hughes, Lee, Lavery, and Shen 2001] that only contain the dependency chain of the critical load or branch.

Helper threads that run on the same code as the main thread have a trivial construction if any since it consists of fetching the same instructions as the main thread. Hand constructed helper threads provide the smallest possible helper thread, lowering overhead, but are impractical due to the human effort needed to interpret profiling results and select the instructions to include in the helper thread. To find a balance between these two methods automatic helper thread construction is used. Helper thread construction differs depending on what type of critical instructions are being addressed. When addressing miss penalties, to identify what loads need helper threads a compiler or memory access simulator is used to evaluate cache performance [Collins, Wang, Tullsen, Hughes, Lee, Lavery, and Shen 2001]. For branch prediction, profiling could be used since no overhead would be added to runtime, but construction during execution was found to provide speedups since the mechanism used is not on the critical path [Chappell, Tseng, Yoaz, and Patt 2002].

Construction of helper threads using a memory access simulator is a straightforward process requiring few steps. During simulation if a load causes a cache miss the instruction 128 instructions prior is marked as a potential trigger. As the simulation continues to execute, whenever the potential trigger instruction is encountered the instruction stream is observed to see if the load that caused the original cache miss misses again. If the load does not cause a cache miss the instruction is removed from the list of potential triggers. If the load is observed as a consistent problem load, the

instructions between the trigger instruction and the critical load are marked as potential instructions for a helper thread. Once instructions unnecessary for address calculation are removed a helper thread of 5 to 15 instructions in size remains [Collins et. al 2002].

Constructing helper threads during execution to address difficult branches has a similar procedure with slightly greater complexity. Crucial to branch difficulty is the context in which a branch occurs. Taking the context of a branch into account makes these helper threads more efficient since the helper thread will only be used when the branch is truly difficult to predict. However, this addition of context increases the complexity of the helper thread construction and the associated hardware (figure 1). This system uses a "path", the previous n taken branches, to define a context. If the branch at the end of a path is found to be consistently difficult to predict, the promotion logic signals for the instructions from the post-retirement buffer to be copied into the scanner (512 instructions in the simulations). The scanner removes any instructions irrelevant to the branch, and puts the remaining instructions into the "microthread" construction buffer. As a last step, an instruction is inserted at the end of the helper thread to communicate the branch outcome generated by the helper thread back to the front-end of the machine. Once the helper thread is constructed it is stored into the MicroRAM, which will be addressed in section 2.3. [Chappell, Tseng, Yoaz, and Patt 2002]

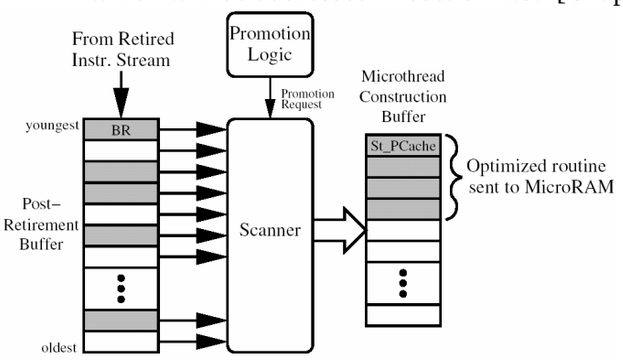

Figure 1. The helper thread construction hardware. Microthread is simply the name given to the helper thread [Chappell, Tseng, Yoaz, and Patt 2002]

2.2 Spawn Point Selection

Critical to speedup provided by helper threads is whether or not the critical information is computed in time, allowing for branch resolution prior to the main thread reaching the branch or data to begin transferring from memory to the cache. Obviously, this is not an issue for sequential helper threads since the helper can be run until the critical steps have been executed. However, helper threads executed at the same time as the main thread depend heavily on timeliness of computation to elicit a speedup. One area that directly contributes to this is the location in the main thread that spawns the helper thread or signals it to begin executing (Figure 2). It is critical to start the helper thread as early as possible so the information is available prior to the main thread reaching the critical section. However, to lessen overhead of helper thread execution two issues are critical; the helper thread must be as small as possible and the main thread must not deviate from the control flow used to define the helper thread. To satisfy these two requirements and thus lessen overhead, a spawn point near the critical instruction is required, resulting in fewer instructions in the helper thread and a greater probability that

the problem instruction will be executed. The criteria for less overhead and early spawn time are orthogonal, and a balance between the two is used.

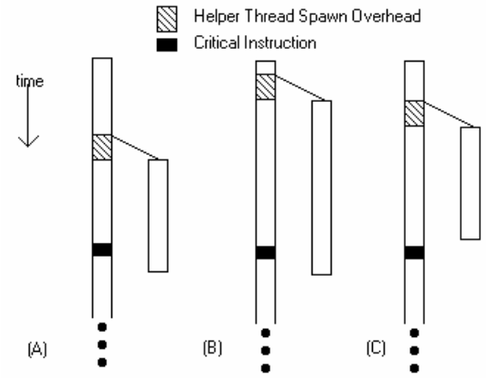

Figure 2. Spawn Point Selection. The main thread is the left bar, and the helper thread is the right bar. (A) The spawn point is too close to the critical instruction. (B) The spawn point is too far away, increasing the number of instructions in the helper thread. (C) The "sweet spot" was used, allowing for the helper thread's result to be used.

Automatically constructed helper threads do not have to worry about spawn point selection. The trigger instructions defined by the helper thread construction methods are simply used as spawn points and whenever the main thread fetches, renames, or issues one of those instructions the corresponding helper thread begins execution. For hand constructed helper threads the researchers' choose instructions to act as trigger instructions or inserted explicit spawn instructions, which were additions to the ISA, in certain "'sweet spots' where the latency tolerance is maximized for a given" helper thread [Zilles and Sohi 2001].

The majority of papers disallow helper threads to spawn other helper threads, but one work utilized this concept to address problem instructions within loops. Rather than having the loop simply contained within the helper thread, loop carried dependences are calculated, another helper thread is spawned and the information is passed to it, and the helper thread continues executing as normal. Research found this method of "chaining triggers" to provide speedups [Collins, Wang, Tullsen, Hughes, Lee, Lavery, and Shen 2001], however methods of software pipelining seem to be a sensible alternative to this method, which would flood the processor with many more threads than available contexts or produce results too quickly. With results available too soon, prefetched cache lines may be displaced before the main thread attempts to load them, requiring more complicated helper thread control mechanisms.

2.3 Stopping the Helper Thread

To lessen overhead and execution interference caused by helper threads it is critical to terminate a thread when its results will no longer be useful. A helper threads results are no longer useful whenever the main thread deviates from the assumed control flow of the helper thread or when the main thread has caught up to helper thread.

Different control flow is only an issue if branches are removed from helper threads. This is done to lessen the complexity of the hardware needed for helper thread execution and is not an issue for helper threads on SMT's which can use the normal branch predictor. To make sure the helper thread being executed is following the correct control flow each helper thread has a vector describing its branch history, and whenever the main thread deviates from that history the helper thread is terminated. Another instance where control flow can deviate from the main thread is when a loop is contained in a helper thread. To reduce the number of instructions that are in a helper thread the

exact loop exit condition may be replaced with a simple "for" loop if it requires multiple instructions to test. In this case the loop is set to iterate for a predefined number of times or until a memory error occurs, such as when traversing a linked list and the end is reached, resulting in a null pointer lookup.

For most helper threads, the main thread cannot catch up since the helper thread contains only a small subset of the total instructions of the main thread. However, when the full code is used as a helper thread it is critical to know when the main thread has caught up. Knowing when the main thread has caught up to a helper thread of this type is difficult since different control flow paths may occur. To know when the main thread has passed the helper thread the hardware records the PC of the first instruction executed by the helper thread and the total number of instructions executed by the helper thread. If the main thread has executed more instructions since encountering the start PC of the helper thread, then the helper thread is terminated. This method is not perfect, but is easy to implement and increases efficiency [Luk 2001].

2.3 Where Helper Threads are Stored

The issue of the overhead of helper thread execution is obviously crucial to performance, and one major cause of slowdown is the resources stolen from the main thread. One way in which helper threads interfere with the main thread is through instruction fetch bandwidth. Most researches simply saw this as an unavoidable fact of helper threads and appended the code for helper threads to the end of the executable, unless the helper thread was a copy of the main thread. Other works decided to make the trade off of increasing processor complexity to minimize this fetch overhead. To do this "MicroRAM" [Chappell, Stark, Kim, Reinhardt, and Patt 1999] is used. MicroRAM is a small content addressable memory on chip that stores the code for the helper thread in the internal ISA of the chip. This approach eliminates any decrease in the main thread's fetch bandwidth, but adds to the complexity of the issue stage of the pipeline, requiring the chip to accept inputs from both the main thread and the helper thread. How this is handled is discussed in section 3.2.2.

3. Parallel Helper Thread Implementations

3.1 SMT Based Implementations

The paradigm of SMT processors greatly decreases the complexity of helper thread implementations by allowing multiple threads to execute simultaneously without corrupting each other's states. This capability not only speeds up the system as a whole when multiple processes are running at one time, but also allows helper threads to execute without worries of corrupting the main thread.

3.1.2 Communicating Results to the Main Thread

Threads executing on an SMT are not totally independent since there is a single shared cache for multiple threads of execution [Collins, Wang, Tullsen, Hughes, Lee, Lavery, and Shen 2001]. However, for helper threads this simplifies prefetching. Whenever a helper thread is spawned to prefetch a load the data is simply fetched by the helper thread and put into the shared cache, requiring no explicit communication of results. This also brings up the crucial point that since the helper thread's memory is shared with the main thread, helper threads can not store values in memory, or else the main thread's state would be corrupted.

Helper threads used to make branch predictions require extra hardware since there is no structure built into SMT processors to correlate predictions to branches. To make use of a helper thread's prediction a "branch correlator" is used (Figure 3). The correlator consists of several queues that store the future predictions for branches tagged with the PC of the branch it is used to predict. To handle mispredictions and late predictions an entry in a queue is allocated and marked every time a branch is encountered, and deallocated when a branch is resolved.

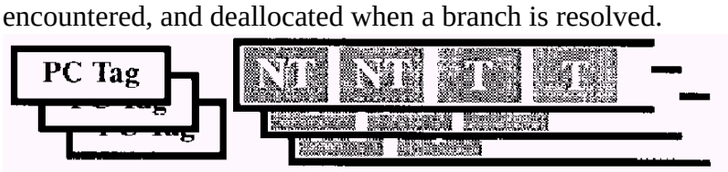

Figure 3. The Branch Correlator [Zilles and Sohi 2001]

3.1.3 Communicating Values to the Helper Thread

One major problem with implementing helper threads on SMT processors is getting initial values into the registers for the helper thread. This issue of communicating values to the helper thread may constitute a major portion of thread spawn overhead. To minimize overhead, but also curb the amount of extra hardware or functionality needed several different methods have been proposed to communicate values from the main thread to helper threads at spawn time. The fastest methods work by copying the mappings from the main thread's register rename table. Two major issues arise from this methodology. One issue is the interference that may occur since rename ports are being used by the helper thread while the main thread may need them. To limit this interference it is important to know that the number of values that the helper thread requires is limited and not all rename ports are needed by the main thread while the helper thread is being spawned [Zilles and Sohi 2001]. The second issue is that the main thread may free and subsequently overwrite a physical register before a helper thread is done with it. To prevent the main thread from overwriting values crucial to helper threads, reference counts are used. Whenever a helper thread copies the mapping of a register, the reference count for that register is incremented, and when a thread frees the register the reference count is decremented. When the reference count is zero (the last thread using the register frees it), the register is placed back into the free pool.

Software based communications take longer, but benefit from not needing to modify the processor. Researches working on already existing processors simply suggested writing all of the important register values to memory and having the helper thread read the values from there. One problem with this is that, if the helper thread requires a large number of values the spawn latency can significantly degrade performance. Due to the limited number of required values only a small latency occurs as a result of this method [Luk 2001]. Researches working with Intel had the opportunity to leverage a research processor, and its on chip buffers. Members of the Itanium processor family have buffers that are architecturally visible and accessible by all thread contexts to temporarily store spilled registers [Collins, Wang, Tullsen, Hughes, Lee, Lavery, and Shen 2001]. The "Live-in Buffer" is a portion of these buffers allocated specifically to allow communication between parent and child threads. The issue of

latency as a result of too many values is still present in this methodology, but transfer of data is faster and there is a zero percent chance of critical values being displaced since the cache is not used.

3.2 Superscalar Based Implementations

One issue never adequately addressed in papers, which based their research on SMT based helper threads, is how a real system load would affect their performance. Some works experimented with varying the number of contexts supported by the SMT processor, and it was found that with more contexts available for helper threads performance improved (figure 4). However, for a real system more than one normal thread could be ready to execute at a time, limiting the number of spare contexts available for helper threads. To address this problem a superscalar processor was altered to implement helper threads, allowing the maximum number of helper threads to be dependent on the hardware, not the system load.

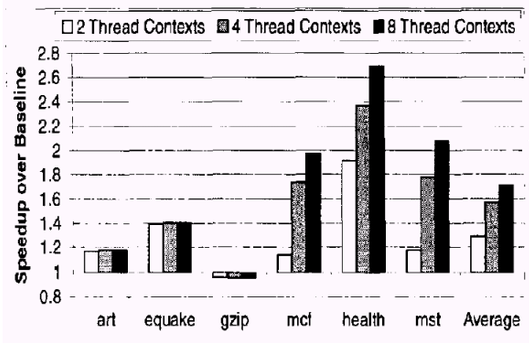

Figure 4. The affect of more SMT contexts on helper threads.

3.2.1 Additions to the Superscalar Processor

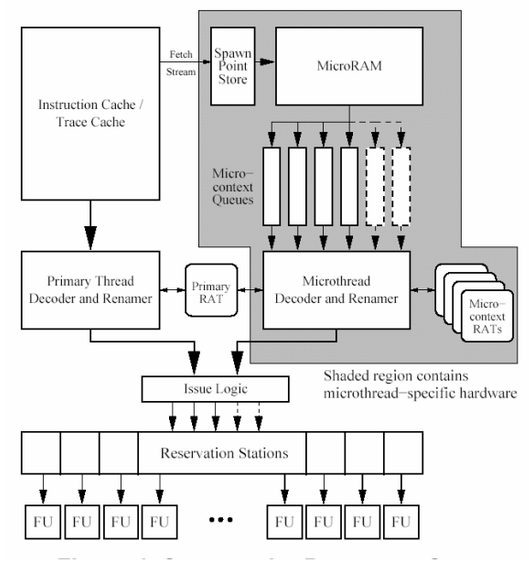

Figure 5. The Altered Processor Core [Chappell, Tseng, Yoaz, and Patt 2002].

The additions to the processor are mostly self-explanatory, and some result in less overhead when compared to SMT implementations of helper threads. The "Spawn Point Store" provides a mapping from the PC of main thread instructions to the helper threads that need to be executed when that instruction is fetched. "MicroRAM" stores the code for the helper threads, and prevents the helper threads from interfering with the main thread's fetch bandwidth. "Microcontext Queues" are temporary storage for helper thread code while earlier instructions in the helper thread are decoded and renamed. To communicate values from the main thread to the helper threads, ports were added to the checkpointed state of the register alias table, and register mappings are simply copied to the "Microcontext RATs". Since checkpointed states are used rather than the current RAT, this prevents interference with the main thread's execution. However, to make this work helper thread spawn points must be on checkpoint boundaries, or spawn instructions could cause the RAT to be checkpointed.

3.2.2 Alterations to the Superscalar Processor

Once the instructions for the helper thread are decoded and renamed they are sent to the same issue logic as the main thread. To handle this the issue logic must be altered so that it accepts inputs from both the main thread and the helper thread hardware, without starving either one. To deal with this a multiplexer is placed on each issue slot, and a helper thread instruction is issued whenever there is no instruction provided by the main thread. However, to prevent starvation of helper threads, after a cycle for which every issue slot is used by the main thread, a cycle of only helper thread instructions is issued. Another scheme rotating between exclusive helper thread and exclusive main thread issuing cycles were tested, but that scheme results in less efficient use of resources [Chappell, Tseng, Yoaz, and Patt 2002 Micro].

To prevent changes to the state of the main thread, helper thread instructions must "fall off the end" of the execution pipeline. This prevents any changes being made to the main thread's register state, reorder buffer, or memory system. To make prefetching by helper threads useful the memory context is simply shared between the threads, but all memory exceptions caused by the helper thread are suppressed.

3.2.3 Advantages of this Implementation

Besides for less interference with the main thread of execution due to internal storage and checkpointed RAT copying, superscalar implementations of helper threads provide other benefits over SMT based helper threads. Unlike SMT based implementations of helper threads, the modified processor can be easily scaled to provide more helper thread contexts. By simply adding more microcontext queues, increasing the size of the MicroRAM, and modifying the issue logic for better instruction bandwidth (issue width is constant) the processor will be able to support more helper threads with a minimal increase in complexity. To further improve performance extra read ports may be added to the checkpointed RAT or more decode/rename slots may be connected to the microcontext queues as the processor is scaled. Rather than adding full contexts to the processor, which drastically increases complexity and cost, the processor can be easily scaled to fit the budget and demands of a consumer.

4. Sequential Helper Thread Implementations

Sequential helper threads work in a manner similar to "Slipstream Processors" [Purser, Sundaramoorthy, and Rotenberg 2000]. However, rather than constantly executing a copy of the program on a separate context, an exact copy of the program is run as the "future thread" [Balasubramonian, Dwarkadas, and Albonesi 2001] on the processor whenever the main thread runs out of its allocated set of registers, signifying limited instruction level parallelism. To allow the future thread to make progress while the main thread is stalled, a time out mechanism is used. Whenever an instruction has not been issued for over 30 cycles, corresponding to an L2 miss, its registers are recycled and all of its dependent instructions are also timed out. This allows more efficient use of registers and issue slots, providing faster execution since the shortened program runs as though all memory references hit in the caches. Once the main thread begins to free registers it can continue to execute using information collected by the future thread to make faster progress.

4.1.1 How the Future Thread Helps the Main Thread

While executing, the future thread can resolve branches, prefetch data, and calculate values that will be reused by the main thread. For the main thread to access the results of the future thread extra hardware needs to be added to the processor (figure 6)

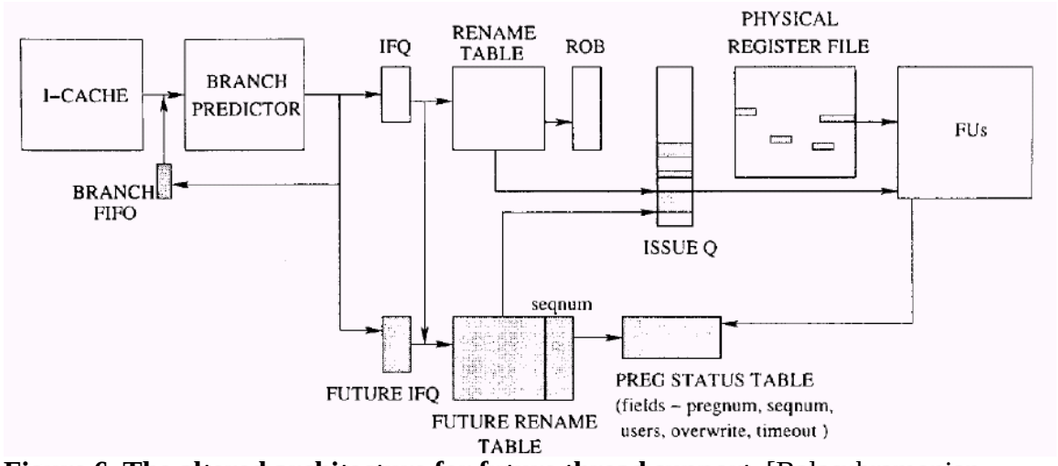

Figure 6. The altered architecture for future thread support. [Balasubramonian, Dwarkadas, and Albonesi 2001]

Branch prediction communication and prefetching are handled simply by this architecture. Since the processor shares the cache for both the main thread and the future thread, any loads performed by the future thread will affect the main thread. This also brings up the crucial point that the future thread cannot store any values to the cache or memory or else the main thread will be corrupted. Corruption cannot occur with registers since mutually exclusive subsets of the register file are used for each thread. A FIFO queue is used to communicate branch predictions to the front end.

Value reuse is more complicated and relies on the "Preg Status Table" and the future rename table. When the main thread dispatches an instruction, a lookup is performed in the status table using the instruction's sequence number. If a valid mapping

still exists it is copied from the future rename table and the instruction does not need to be executed. If the mapping does not exist, then the future rename table is updated so that the future thread will have the correct value when it executes.

4.1.2 Selecting When to Run the Future Thread

This implementation of helper threads does not target any specific critical loads or branches, but is simply triggered when the main thread has used up all of its allocated physical registers. For this reason for certain programs or during certain phases of program execution, the future thread may hurt performance. To handle this the partitioning of registers between the main thread and the future thread is done dynamically. To do this for every 100k instructions hardware counters that keep track of branches and L1 cache misses are examined. If there is a significant change in these values from the previous interval then it is assumed the program phase has changed and the next several intervals are used as an exploration phase with each interval having different register partitions. After the exploration phase, the partition with the highest IPC is used as the new partition until another change in phase is encountered. The overhead of this technique is insignificant or is cancelled out by the improvement in program performance as is seen in figure 7.

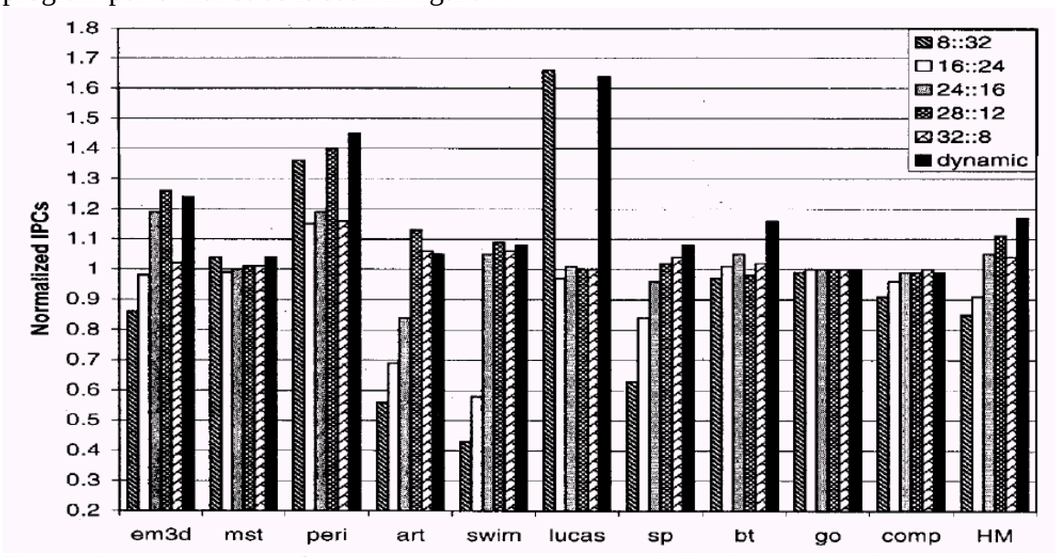

Figure 7. Comparison of static register partitions and the dynamic technique used with the future thread.

5. Conclusions:

This paper has discussed helper threads and their ability to improve cache performance and branch prediction using several different implementations with the major implementations being parallel helper threads and sequential helper threads. With each paradigm offering different advantages and disadvantages.

Parallel helper threads are used to target specific problem instructions such as cache missing loads or difficult to predict branches. For this reason the techniques used to define these problem instructions is critical to the speedup provided by the helper threads. Also critical to the performance improvements made by helper threads is the

overhead they incur and whether or not the results are provided in time to help the main thread. To lessen overhead and still allow for timely execution two hardware platforms have been proposed; SMT processors and adapted superscalar processors. The SMT processor takes advantage of already existent hardware to execute independent contexts, but the communication needed to provide starting values for the helper thread and interference with execution and fetch bandwidth result in significant overhead. Adapted superscalar processors have storage on chip and instruction issue heuristics to maintain main thread fetch bandwidth and limit execution overhead of helper threads. One disadvantage of the adapted superscalar approach is that it requires significant alterations, which may be too costly if the processor works with programs that do benefit from helper threads. However, unlike SMT processors the adapted processor easily scales to allow more helper threads to execute.

Sequential helper threads do not target specific problem instructions but improve cache performance and branch prediction whenever there is limited ILP for the main thread. By executing only when the main thread would be stalled and allowing for near perfect branch prediction, prefetching, and calculation reuse, sequential helper threads have very little overhead while still improving performance. The major drawback of this approach is that by partitioning the register set into future thread and main thread registers, programs with sufficient ILP to utilize the full register set may suffer. However, to address this issue the register set is dynamically partitioned, allowing for a register partitioning that favors the current phase of execution.

References

R. Balasubramonian, S. Dwarkadas, D. Albonesi. "Dynamically Allocating Processor Resources Between Nearby and Distant ILP", *28th International Symposium on Computer Architecture*, (ISCA '01), Jul. 2001.

R. Chappell, F. Tseng, A. Yoaz, and Y. Patt. "Difficult-Path Branch Prediction Using Subordinate Microthreads", *Proceedings of the 29th Annual International Symposium on Computer Architecture*, (ISCA '02), May 2002.

R. Chappell, F. Tseng, A. Yoaz, and Y. Patt. "Microarchitectural Support for Precomputation Microthreads", *35th International Symposium on Microarchitecture*, (Micro-35), Nov. 2002.

J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen. "Speculative Procomputation: Long-range Prefetching of Delinquent Loads", *Proceedings of the 28h Annual International Symposium on Computer Architecture*, (ISCA '01), Jul. 2001.

C.-K. Luk. "Tolerating Memory Latency Through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors", *Proceedings of the 28th Annual International Symposium on Computer Architecture*, (ISCA '01), Jul. 2001.

Z. Purser, K. Sundaramoorthy, and E. Rotenberg. "A Study of Slipstream Processors", *33rd International Symposium on Microarchitecture* (MICRO-33), Dec. 2000.

C. Zilles, G. Sohi. "Execution-based Prediction Using Speculative Slices", *Proceedings of the 28h Annual International Symposium on Computer Architecture*, (ISCA '01), Jul. 2001.