Code Semantic-Aware Runahead Threads

Tanausú Ramírez[1], Alex Pajuelo[1], Oliverio J. Santana[2], Mateo Valero[1,3]

[1]Universitat Politècnica de Catalunya, Spain. tramirez,mpajuelo@ac.upc.edu.

[2]Universidad de Las Palmas de Gran Canaria, Spain. ojsantana@dis.ulpgc.es

[3]Barcelona Supercomputing Center, Spain. mateo.valero@bsc.es

Abstract

Memory-intensive threads can hoard shared resources without making progress on a multithreading processor (SMT), thereby hindering the overall system performance. A recent promising solution to overcome this important problem in SMT processors is Runahead Threads (RaT). RaT employs runahead execution to allow a thread to speculatively execute instructions and prefetch data instead of stalling for a long-latency load. The main advantage of this mechanism is that it exploits memory-level parallelism under long latency loads without clogging up shared resources. As a result, RaT improves the overall processor performance reducing the resource contention among threads.

In this paper, we propose simple code semantic based techniques to increase RaT efficiency. Our proposals are based on analyzing the prefetch opportunities (usefulness) of loops and subroutines during runahead thread executions. We dynamically analyze these particular program structures to detect when it is useful or not to control the runahead thread execution. By means of this dynamic information, the proposed techniques make a control decision either to avoid or to stall the loop or subroutine execution in runahead threads. Our experimental results show that our best proposal significantly reduces the speculative instruction execution (33% on average) while maintaining and, even improving the performance of RaT (up to 3%) in some cases.

1 Introduction

Current processor trends focus their designs on Multithreading [16][19] and Multicore [10] architectures. However, the ability to exploit more performance (both instruction level parallelism and thread level parallelism) in these processors still hits the memory wall problem [18]. Long memory latencies continue having importance for multithreading and multicore processors due to their impact on performance. For instance, a thread with a pending long-latency load will not make forward progress, and even worse, it will hoard shared resources starving the other threads and harming the overall system performance.

Several researches identified the problem and proposed different long-latency load aware policies [2][3][15]. These proposals focus their actions on limiting the execution or the amount of resources allocated by memory-intensive threads. When a long-latency load is detected, these techniques either stall or flush the offending thread in order to avoid fetching more instructions and holding more resources. A drawback of these SMT policies is that they are not able to exploit the available memory-level parallelism (MLP), since the memory-intensive threads are stalled.

One current appealing solution in this sense is Runahead Threads (RaT) [11]. RaT detects when a thread executes a long-latency load, and then it turns the offending thread into a *runahead thread* to expose MLP through prefetching. By using RaT, SMT processors both improve single-thread performance due to the fact that the runahead thread prefetches data under long-latency loads and reduce resource clogging since the runahead thread recycles faster the shared resources instead of holding them. However, these benefits can be spoiled if the thread executes a large number of useless speculative instructions without doing prefetch.

The objective of this work is to limit this useless extra work through code semantic analysis for improving runahead thread efficiency. This efficiency improvement consists of reducing the number of speculative instructions executed in parts of the programs while keeping the optimal generation of useful prefetches (and hence the performance) of a runahead thread. We propose simple control mechanisms that detect particular code semantic structures for overseeing their efficiency in order to select the useful executions and to discard the useless ones. The key point of our proposals is to obtain dynamic information about these structures, such as loops and subroutines, to decide whether they are useful or not during runahead thread execution. Based on this information, we take a control action (1) to skip the execution of useless parts of the program or (2) to stall the runahead thread execution to avoid unnecessary speculative execution.

The different experiments show that stalling a useless loop or subroutine execution presents better results from the efficiency point of view. Our best approach reduces 33% the number of speculative executed instructions of runahead threads. In addition, these techniques achieve a comparable performance related to the original RaT mechanism.

2 Runahead Threads

Runahead Threads (RaT) [11] is a mechanism to alleviate the problems related to long-latency loads in SMT processors. RaT focuses its solution on exploiting MLP by applying Runahead execution on SMT processors. *Runahead* execution [9] is a mechanism whose goal is bringing speculatively data and instructions into the caches without blocking the processor. When a thread undergoes a long-latency load, and this load reaches the head of the reorder buffer, the thread is turned into runahead mode. RaT takes a thread context checkpoint and the thread enters into a speculative execution until the load is serviced. Once the long-latency load returns from memory, the thread context state is restored from the checkpoint and it resumes its normal execution. With this ability, *RaT* improves single-thread performance by exploiting the available memory level parallelism since allows memory-intensive threads to speculatively going in advance doing prefetch. At the same time, RaT improves overall system performance, since runahead threads do not clog up shared resources (and thus they do not starve other threads) on long-latency load misses. The runahead thread allocates and deallocates shared processor resources quickly instead of holding them until the long-latency load completes. RaT provides better SMT performance and fairness than prior SMT resource-aware policies.

However, RaT has no information about the existence of available prefetches for a particular runahead thread. Whenever a runahead thread executes without doing prefetching, it performs useless extra work. Therefore, this thread does not contribute to improve the performance but performs useless speculative work. Based on our studies, RaT increases the number of executed instructions by 52% for 2-thread workloads compared to ICOUNT [16], and 46% for 4-thread workloads on average.

3 Overseeing Program Structures to Improve RaT Efficiency

In this paper, we address how to control RaT useless extra work through code semantic analysis to reduce the number of speculative instructions executed and to improve runahead threads efficiency. We investigate several mechanisms to analyze the usefulness of particular program structures during runahead thread execution. We devise these mechanisms with two main features in mind. First, the code semantic patterns should be easy to detect in order to design low-complexity mechanisms. The new structures or components for their implementation should not complicate more the hardware to keep RaT a feasible mechanism. Second, these code structures should have a large number of instructions involved, that is, the highest possible granularity of speculative instructions to capture a big ratio of extra work. Therefore, we choose two semantic patterns very common in the program codes which fulfill the requested features:

- *the loops*, which basically are repetitive sequence of instructions very common in the programs.
- *the subroutines*, groups of code instructions that performs a specific task and are called frequently inside the program.

Our goal is to oversee when a runahead thread is executing a useful loop or subroutine. On the contrary, we take a decision to increase the efficiency of RaT. We consider a loop as *useful* when the current runahead thread can issue at least one *useful* load to the memory system during each loop iteration. Likewise, we define *useful load* as a valid runahead load which misses at the first level cache, and therefore can prefetch data from upper memory levels. We also extend the previous *useful work* concept to subroutines. That is, we consider a subroutine as useful if during its execution the runahead thread issues at least one useful load.

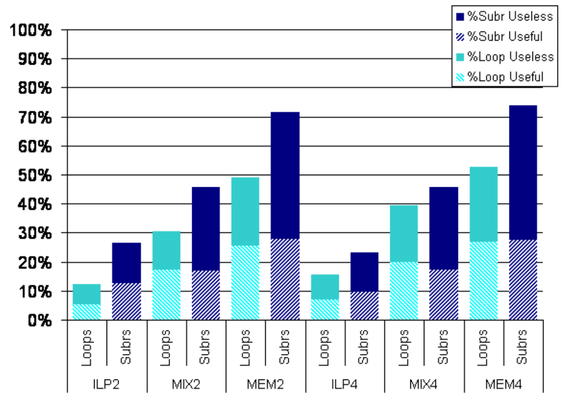


Figure 1. Loop and Subroutine stats in Runahead Threads

In Figure 1 we show the ratio of loops and subroutines detected in runahead threads regarding the total loops and subroutines respectively executed during the program execution. Each bar is splitted into the percentage of useless (plain part) and useful (striped part) loops or subroutines for each kind of workloads (see details of our framework in section 4). As the Figure shows, the tendency of usefulness for both code structure executions is similar, with an average 50% of loops and 41% of subroutines. Thus, if we detect and discard the other percentage of loops or subroutines which do not contain misses (50% and 59% respectively) in runahead threads, we can improve the RaT efficiency. Next, we describe deeper the functionality and implementation for each of the approaches proposed for overseeing both loops and subroutines to control the useless runahead thread executions.

3.1 Loop Control

The first approach is based on controlling the loop execution during runahead mode. The idea is to detect when a runahead thread is inside a loop, and then to analyze the loop execution to find out that the runahead thread is really doing "useful work". Finally, depending on the usefulness of the speculative loop execution, our mechanism does a control action: it forces the exit of the loop and to continue the execution after that, or it directly stalls the runahead execution.

The loop usefulness control techniques we proposed consist of three parts. First, we need to identify a loop during the runahead execution. Second, once the loop is detected, we need to determine if this loop is useful for the runahead thread. Third, we need to take a control decision based on the loop usefulness. The first two steps are common for each of techniques, since is the procedure to detect the usefulness of the loops. The last step is different depending on the action to take, which fixes a different purpose for each particular case.

1.- Loop Detection. In order to simplify the design of our proposal, we just take into account simple loop structures made by the compiler [1]. That is, the dynamic execution of a simple loop implies the repetitive execution (loop iteration) of the same group of instructions (loop body) which are enclosed between a backward branch and its target [4][5]. The backward branch is considered the loop end (last instruction) and its target address is considered the first instruction of the loop body.

Therefore, when a backward branch is predicted as taken during a runahead thread execution, our mechanism starts the loop detection process. To do this, we store the branch program counter and its instruction target in two registers, `LOOP END` and `LOOP START` respectively. We need this information to control when the runahead thread is executing inside the range of the loop body. If the same backward branch is found and it is taken again, then we identify it as a loop branch. We indicate this state to the runahead thread using a flag (`INLOOP`), which is set to one. In figure 2 we illustrate the loop detection overview. This Figure shows an example of a loop structure that is detected by our mechanism. The loop body starts at instruction @20 and finalizes with the loop branch @44, which are stored in the `LOOP START` and `LOOP END` registers respectively as it is showed in the figure.

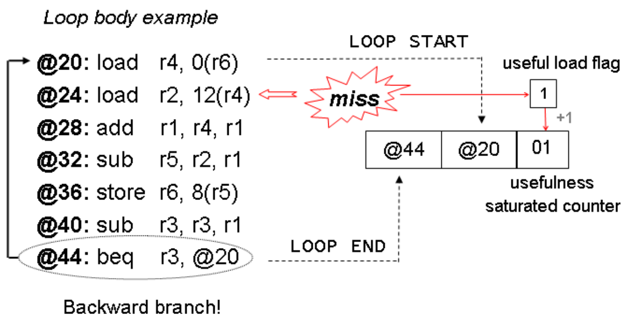


Figure 2. Loop control mechanism example

2.- Loop Usefulness Check. Once our mechanism confirms the existence of a loop, it is ready to oversee the loop usefulness. We introduce a *useful load flag* to capture the useful work done during each loop iteration. This flag is set to zero at the beginning of the next iteration of the loop. From this point onwards, this flag is set to one when there is a runahead load that misses in the first level data cache and initialized to zero each time the backward branch is taken (that is, every new iteration). If the value is zero each time we find the loop branch, this means there were not useful loads in the previous iteration and, then, the loop is candidate to be considered as useless. The advantage of our mechanism is that this usefulness test is made during the own loop execution. That is, we use current dynamic information while the runahead thread is inside the detected loop.

3.- Loop Control Decision. Thanks to the previous analysis we can select which loops are useful during a runahead execution. On the contrary, if we detect a loop iteration as useless, the mechanism makes a control decision. In this sense, we propose two different control actions associated to the loop control mechanism to improve the RaT efficiency: one version focuses on performance and the other on energy consumption. The first one is *Loop Reverse -LR* which forces the exit of the useless loop by not taking the end-loop branch. This allows runahead thread to continue the speculative execution beyond this loop trying to capture further prefetching for improving performance. The second one is *Loop Stall -LS*, which simply stalls the runahead thread in order to avoid to continue executing useless speculative instructions in the loops. In this case, the goal is to directly cut down the extra work, thereby reducing the energy consumption.

[1]Compaq C and FORTRAN Alpha compilers compile most for, while, and do loops with the conditional branch check at the bottom of the loop with a negative offset.

With the described implementation, our mechanism is able to perform the control action (according to LR or LS technique) when the loop branch is found three consecutive times in a runahead thread execution. In this point, the corresponding technique contains information about the usefulness of the previous iteration of the loop. This depends on whether the *useful load flag* is set (useful) or not (useless). Although the control action could be done directly using only this flag, we explore an additional level of confidence by means of introducing a saturated counter of two bits. The idea is to tune the decision with regard to the different loop body structures, in which some loops contain different execution paths that produce useful and useless iterations alternatively. Thus, we can take into account the usefulness of control-dependent loads under hard-to-predict branches inside the loops.

This saturated counter is incremented for each iteration with the *useful load flag* equal to one (as we show in the example of Figure 2 due to the @24 instruction miss), and it is decremented otherwise. With this modification, our mechanism performs the control action when the saturated counter reaches zero the next time the backward branch is taken. Each action represents a different version of the loop control mechanism: (1) Loop Reverse -exit the loop or (2) Loop Stall -stall the execution. According to our studies, the saturated counter provides better results than making a direct action. We will evaluate the loop control mechanism versions LR and LS in Section 5.

3.2 Subroutine Control

The second approach on the line of code semantic control to improve the RaT efficiency focuses on the subroutines. The idea is the same that previous section for loops, but applying the *useful work* concept to subroutines. We refer *subroutine* to all the so-called procedure, method or function at the high-level languages which performs a specific task in a program. A subroutine is a piece of code so that it can be started (called) several times and/or from several places during a single execution of the program, including from other subroutines, and branch back (return) to the point after the call once its task is done. This feature makes subroutine a good code structure choice to focus on controlling a high granularity of speculative instructions per runahead thread. Like the control loop mechanism, the subroutine control mechanism consists of three steps: the subroutine detection, the usefulness analysis and, finally the control decision in function of the technique applied.

1.- Subroutine Detection. A language's compiler usually translates subroutine calls and returns into machine instructions. So, the subroutine detection is straightforward in this case, since we simply capture the subroutine call at the decode stage once the corresponding instruction is identified (e.g. in the Alpha ISA these are jsr or bsr). The end of the subroutine execution is delimited by the corresponding return assembler instruction (Most ISAs provide these kinds of instructions. So, our techniques can be easily ported to any processor). Each time a subroutine call is detected, the program counter of the subroutine instruction is stored in a new register (called `INSUBROUTINE`), which, at the same time, indicates the runahead thread is in a subroutine.

2.- Subroutine Usefulness Check. Although the idea for the usefulness is the same as loops, in this case the mechanism should do a deeper analysis. If inside a subroutine body another subroutine is called, we consider that the usefulness information of the called subroutine also belongs to the caller subroutine. So, it is needed to track the usefulness through the nested subroutine calls. In this sense, we need to control the deep level of a subroutine between the parent and the child calls. In addition, we need to propagate the usefulness information from the child subroutines to the parents.

For this purpose, we can easily implement this procedure using the return address stack (`RAS`) employed for return address prediction presented in current architectures. We extend each entry of the RAS with one bit for the *useful load flag*. Therefore, when a subroutine call is detected and an entry is inserted in the RAS, we also reset its corresponding flag. During the subroutine execution, the *useful load flag* is set when at least one L1-miss load is found (the current subroutine is at the top of the stack). When a return instruction is executed, the processor pops the corresponding entry from the `RAS`. In this point, we also propagate the usefulness information using an OR operation of the associated *useful load flag* value of this ended subroutine with the *useful load flag* of the parent subroutine, which is now at the top of the RAS. Figure 3 shows a snapshot about this procedure for the subroutine control, in which subroutines A, B1 and A2 have been pushed in the RAS. As we show in this example, the flag propagation is done when subroutine A2 is removed from the RAS and its associated flag is transferred to the parent subroutine B1. At the same time, we update the `INSUBROUTINE` register with the current subroutine PC (B1) taken from the RAS. This procedure is carried out during all program execution, both in normal and runahead mode to keep the consistency of subroutine levels.

3.- Subroutine Control Decision. Whereas for loop control we make the control decision dynamically during the loop iterations, in the case of subroutines this decision is based on its past history. That is, we collect information about the current subroutine execu-

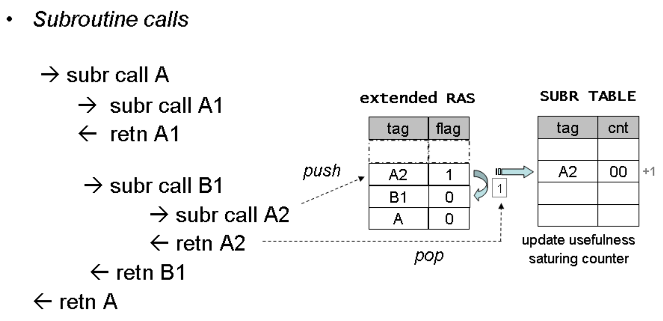


Figure 3. Subroutine control mechanism

tion to decide what to do when this subroutine is called again. Thus, we insert a small table (`SUBR TABLE`), with the same entries as the RAS, to store the subroutine PC and the historical usefulness information of its previous executions. Basically, this consists of the tagPC and an usefulness saturated counter with the same functionality as the one in the loop control mechanism. This table is updated when the subroutine returns, taking the PC from the RAS and updating the saturated counter depending on the *useful load flag* from its corresponding `RAS` entry (see the example in Figure 3 when the subroutine A2 returns).

The `SUBR TABLE` is looked up when a subroutine call instruction is decoded during runahead execution using the instruction PC in order to get the usefulness information from its saturated counter. Based on this information, the different subroutine control techniques make a decision. For this subroutine control mechanism, we propose three different actions when the usefulness saturated counter of a particular subroutine reaches to zero. The first version, *Subroutine Skip -SK*, skips the subroutine execution, ignoring the subroutine call instruction and jumping to the instruction pointed by the return address to continue the runahead execution. Since runahead threads do not modify the memory, the stack of the processor is not modified neither before nor after the subroutine execution. The second one, *Subroutine Stall -SS*, directly stalls the runahead thread execution before doing the subroutine jump once the `SUBR TABLE` have been accessed and detects its useful saturated counter is zero. The third one is a combination of the previous two, in which the technique introduces an additional factor of information to choose one particular action or another. This technique takes profit of the L1miss tracking process to update the useful load flag and extend its functionality to find out if there is available nearby prefetching after a subroutine execution. Depending on this usefulness information, this control subroutine technique decides whether skip the subroutine or stall the execution. We called this last technique ***Post-Subroutine Usefulness -PSU***.

To implement this last version, we simply add one bit more per entry into the `SUBR TABLE` to indicate if there are L1 cache misses after the subroutine execution (*PSU flag*). Besides, we need to remember the PC of the previous subroutine after returning from it to be able to record its post-subroutine usefulness. To do this, we add a new register called `PREV SUBR`. This register is written with the PC of a subroutine that is at the top of the RAS each time there is a return instruction. Then, we check whether this `PREV SUBR` register is not zero when there is a next subroutine call to update the corresponding flag belonging to the previous subroutine. Thus, if there were at least one cache miss after the subroutine execution pointed by the `PREV SUBR` register, its PSU flag in the `SUBR TABLE` is set to one. Otherwise, this flag is reset to zero. Therefore, when applying this PSU technique and the useful saturated counter reaches zero, the runahead thread skips the subroutine execution if the PSU flag that indicates there were L1 misses after this subroutine is set or it stalls the runahead execution if this flag is zero.

4 Experimental Setup

The main configuration parameters of the simulated SMT processor model are listed in Table 1. In this SMT model, we use a dynamic resource partitioning in which the inter-thread sharing of resources is determined by the underlying fetch policy. We use the ICOUNT(2,8) that selects two different threads each cycle (according to their priorities) and fetches up to eight instructions from each thread. Threads coexist in different pipeline stages, sharing the important hardware resources (issue queues, rename registers, functional units, caches, memory system).

Processor core	
Pipeline depth	10 stages
Fetch/decode/issue/execute width	8 way
Reorder buffer size	128 entries per context
INT/FP registers	204 / 204
INT/FP/LS issue queues	64 / 64 / 64
INT/FP/LdSt units	4 / 4 / 2
Branch predictor	Perceptron predictor with 256 perceptrons 4096 x 14 bit local and 40 bit global history
RAS	64 entries
Memory subsystem	
Icache	64 KB, 4-way, 1-cycle access latency
Dcache	64 KB, 4-way, 3-cycle access latency
L2 Cache	1 MB, 8-way, 20-cycle access latency
Cache line size	64 bytes
Main memory latency	300-cycle minimum latency

Table 1. SMT processor model configuration

We use an execution-driven SMT simulator derived from SMTSIM [14] in all of our experiments. We have modified SMTSIM to support simulation checkpoints and a very detailed memory hierarchy modeling bus/port/bank contention, bandwidth, and main memory timings accurately. The experiments have been performed with workloads created by combining single-threaded programs from the SPEC CPU2000 benchmark suite [12]. We consider groups of 2 and 4 SPEC benchmarks to create various workloads of ILP-intensive benchmarks (ILP), memory-intensive benchmarks (MEM), and those consisting of a mixture of the previous two (MIX). All benchmarks were compiled using the Compaq C/C++ compiler with the -O3 optimization level to obtain Alpha ISA binaries. For each benchmark, we simulate 300 million representative instructions using the reference input set. Measurements are then taken using the FAME methodology [17]. FAME re-executes all traces in a multiprogrammed workload until all of them are fairly represented in the final measurements.

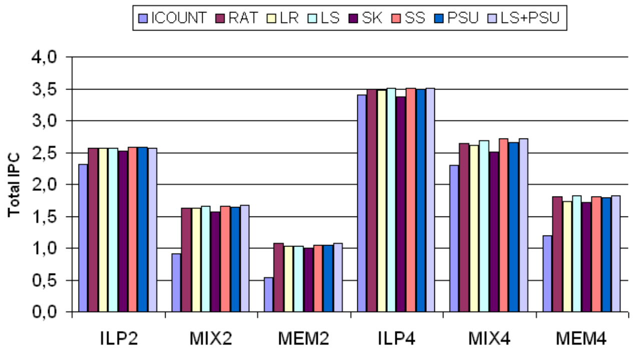
5 Evaluation

In this section, we evaluate the efficiency (in terms of performance and extra work) of the different proposals of this work. We analyze the results for *Loop Reverse* (LR) and *Loop Stall* (LS) for loop control mechanism (section 3.1) and, *Subroutine Skip* (SK), *Subroutine Stall* (SS) and *Post-Subroutine Usefulness* (PSU) for subroutine control techniques (Section 3.2). We also show the combination of the LS with PSU (LS+PSU), since this mix provides the best results among all combinations.

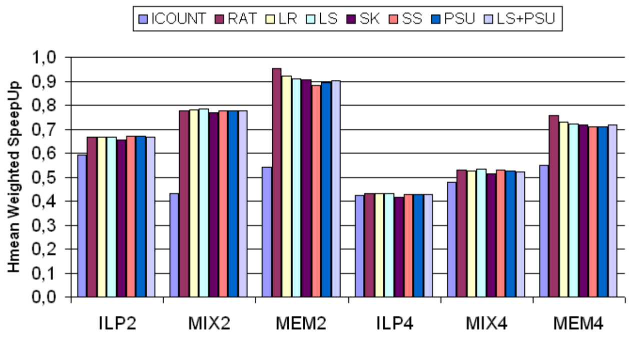
5.1 Performance

We evaluate the performance of our proposals in terms of total throughput and harmonic mean of individual threads speedup (Hmean [7]). Using these two metrics, we show both total system performance and user performance-fairness perception. Figures 4(a) and 4(b) show the throughput and Hmean for the 2-thread and 4-thread workloads grouped by the different sets (ILP, MIX and MEM) respectively. Note, we show results for ICOUNT as the SMT processor baseline and the original RaT proposal for comparison purposes.

It is observed in Figure 4(a) that the performance throughput is very close to RaT for almost all proposed techniques. The “worst” proposal is Subroutine Skip (SK) which only loses a slightly 4% on average compared to RaT. This technique executes more far useless instructions that harms a bit the performance. However, there are other techniques, like Loop Stall (LS) or Subroutine Stall (SS), which achieve throughput improvements over RaT, especially for MIX workloads (SS achieves a 3% improvement in MIX4 workloads). In this case, stalling the useless part of a code execution benefits the other threads that take profit of more available resources. The best result is for LS+PSU combination with a 1% throughput improvement on average compared to RaT.



(a) System Throughput (IPC)



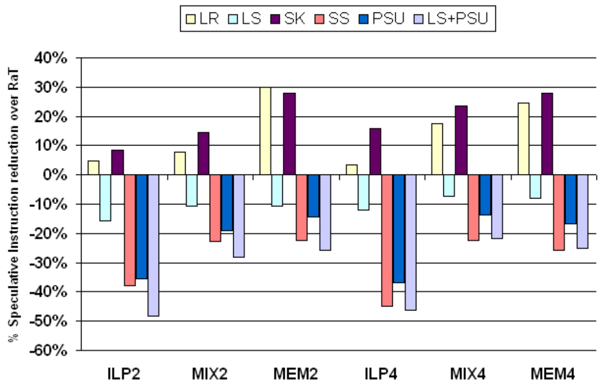
(b) Hmean SpeedUp

Figure 4. Performance

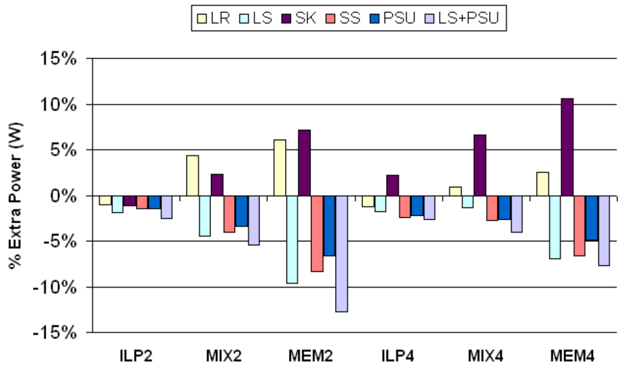
Regarding Hmean, Figure 4(b) also shows that usefulness control techniques get comparable hmean results as RaT. Although there is an overall small hmean reduction for all techniques in memory-intensive workloads (with a maximum of 7% of reduction with SS for MEM2), for the rest of workloads the performance remains the same as with RaT. Even, there is a slight improvement (1.2% on average) for the LS technique in the case of MIX workloads.

5.2 Extra Work

Once we have shown that the proposed techniques obtain similar performance to RaT, the other factor to enhance the RaT efficiency is to reduce the number of executed speculative instructions. Achieving this issue reverts in a net reduction of power consumption. Figure 5(a) shows the percentage of speculative instructions executed by runahead threads when applying our control techniques compared to RaT. As we can observe in this Figure, the results are different among the proposals. While the stall action techniques (LS, SS) and PSU reduce the number of speculative instructions for all workloads (up to 48%), the others LR and SK suffer an increment of 14% and 19% on average respectively. Although LS and SS activate more runahead



(a) % Speculative executed instructions



(b) Average Extra Power

Figure 5. Percentage of Extra work

threads (8% more) due to the stall action that prevents more long latency load from being captured, LR and SK execute a higher number of instructions since they allow runahead threads to execute faraway more instructions after a loop is reversed or a subroutine is skipped. Nevertheless, among the effective techniques, we remark the combination of LS+PSU which achieves 33% reduction on average in the amount of speculative instructions executed with regard to RaT.

We also quantitatively evaluate the power reduction obtained by the different proposals. To measure the power estimation, we include an improved power model based on Wattch [1] in our simulator. We model the power for all the main hardware structures of the processor (functional units, registers, branch predictor, queues, rob, memory system, etc). With this accurate model, we provide results in Figure 5(b) about power reduction compared to RaT for the different mechanisms evaluated. The power results depict a similar trend to instruction reduction results. While LR and SK get a slight power increment (1.9% and 4.5% respectively), the LS+PSU technique achieves an average 6% power reduction compared to RaT.

Finally, to provide insights into how each technique manages the executed runahead instructions according to their different actions, in Figure 6 we show the percentage of reversed or stalled loops for LR and LS techniques, and the percentage of skipped or stalled subroutines for SK and SS. For both control mechanisms, the number of loops or subroutines stalled is lower than those reversed or skipped. The percentage of loops detected as useless is 43% and 23% for reversed and stalled loops whereas in the case of subroutines this percentage is 78% and 64% respectively.

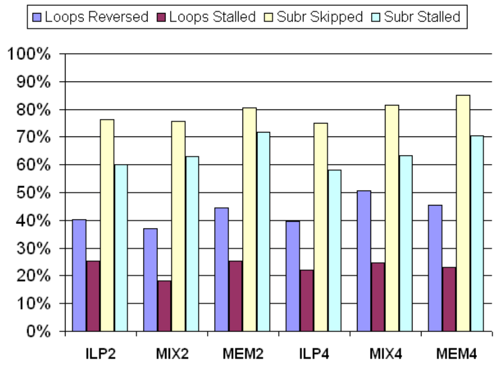


Figure 6. Loops and Subroutines control mechanism statistics.

6 Related Work

Several studies have addressed their researches to enhance subroutines execution and to classify and detect loops. Kaeli *et al.* [6] presented a different Branch History Table (BHT) design that is based on the underlying programming structures. They propose a new stack mechanism for reducing the number of wrong predictions due to the subroutine CALL/RETURN paradigm. By using a set of stacks, the mechanism identifies when a CALL is made and then supplies the correct branch target when the RETURN is encountered. Another work [13] presents a mechanism to dynamically detect the loops that are executed in a program. This technique detects the beginning and the end of the iterations of the loops without compiler/user intervention. Their overall objective is to dynamically obtain coarse grain parallelism by exploiting the speculation of multiple threads of control obtained from a sequential program. In this line, a current research presents the *Loop Processor Architecture* [5], focused on capturing the semantic information of high-level loop structures and using it for optimizing program execution. The LPA design uses a buffer, namely the loop window, to dynamically detect and store the instructions belonging to simple dynamic loops along with all the information needed to build the rename mapping.

Therefore, the loop window can directly feed the execution back-end queues with loop instructions, avoiding the need for using the front-end stages of the normal processor pipeline.

Previous research [8] identified three causes of inefficiency in runahead execution: short, overlapping, and useless runahead periods. This work proposes techniques to improve the efficiency of runahead in single-threaded processors to reduce their occurrence. To eliminate useless runahead periods, they propose a technique with an MLP binary predictor based on two-bit saturating counters. In case there is no MLP to be exploited, a runahead period is not initiated. To eliminate short and overlapped periods, the authors presented different threshold-based heuristics. Although Mutlu *et al.* has the same goal as our work (improving the efficiency of Runahead Threads in this case), we target the problem from a different context. We develop different techniques focused on analyzing dynamically code semantics. We select the useful loops and subroutines and discard the useless ones to improve the RaT efficiency.

7 Conclusions

In this paper, we enhance the efficiency of RaT by a novelty way using semantic program-conscious techniques. Code semantic-aware runahead threads select the useful work to execute analyzing the usefulness of program structures. We propose new simple and effective approaches, both in terms of cost and hardware complexity. Basically, these mechanisms analyze during runahead thread execution the usefulness of loops or subroutines depending on their prefetch opportunities. Using this information, we provide dynamic control techniques that decide either to stall or to skip the loop and subroutine execution.

The best combination of these techniques results in a performance-efficient approach that maintains the performance improvement of RaT while reducing the extra speculative work required (33% less speculative instructions on average) and power consumption (6%).

Acknowledgments

This work has been supported by the Ministry of Science and Technology of Spain under contracts TIN-2007-60625 and CSD2007–00050, the HiPEAC European Network of Excellence and the Barcelona Supercomputing Center (BSC-CNS).

References

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the ISCA-27*, 2000.

[2] F. J. Cazorla, A. Ramirez, E. Fernandez, and M. Valero. Dynamically controlled resource allocation in smt processors. In *Proceedings of MICRO-37*, 2004.

[3] S. Choi and D. Yeung. Learning-based smt processor resource distribution via hill-climbing. In *Proceedings of ISCA-33*, Washington, DC, USA, 2006.

[4] M. R. de Alba and D. R. Kaeli. Runtime predictability of loops. In *Proceedings of the IEEE Intl. Workshop of Workload Characterization (WWC-4)*, Washington, DC, USA, 2001.

[5] A. Garcia, O. J. Santana, E. Fernandez, P. Medina, and M. Valero. LPA: A first approach to the loop processor architecture. In *Proceedings of 3rd HIPEAC Conference*, 2008.

[6] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. *SIGARCH Computer Architecture News*, 19:34–42, 1991.

[7] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In *Proceedings of ISPASS-2001*.

[8] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of ISCA-32*, 2005.

[9] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of HPCA'03*, Washington, DC, USA, 2003.

[10] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30(5):2–11, 1996.

[11] T. Ramirez, A. Pajuelo, O. J. Santana, and M. Valero. Runahead threads to improve smt performance. In *Proceedings of HPCA'08*, Salt Lake City, UT, USA, 2008.

[12] SPEC. Standard performance evaluation corporation (spec) 2000 benchmark suite. http://www.spec.org/cpu2000/.

[13] J. Tubella and A. Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of HPCA'98*, Las Vegas, Nevada.

[14] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Int. Annual Computer Measurement Group Conference*, 1996.

[15] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of MICRO-34*, Washington, DC, USA, 2001.

[16] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of ISCA-23*, NY, USA, 1996.

[17] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. A novel evaluation methodology to obtain fair measurements in multithreaded architectures. In *Workshop on Modeling, Benchmarking and Simulation*, 2006.

[18] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[19] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Proceedings of PACT'95*, Manchester, UK, 1995.