An online scheduler for hardware accelerators on general-purpose operating systems

David Sheldon, Alessandro Forin
Microsoft Research

April 2009
Revised
December 2010

Technical Report
MSR-TR-2009-41

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

An online scheduler for hardware accelerators on general-purpose operating systems

David Sheldon, Alessandro Forin
Microsoft Research

Abstract

This paper presents an online scheduling algorithm for hardware accelerators and its implementation on the NetBSD operating system. The scheduler uses the current performance characteristics of the accelerators to select which accelerators to load and unload. The evaluation on a number of workloads shows that the scheduler is typically within 20% of the optimal schedule computed offline. The hardware support consists of simple cost-benefit indicators, usable for any online scheduling algorithm. The NetBSD modifications consist primarily in loadable kernel modules, with minimal changes to the operating system itself. The measured overhead is negligible when accelerators are not in use, and otherwise scales linearly by a small constant with the number of active accelerators.

1 Introduction

Single-threaded performance has been linked to an ever-increasing clock frequency, but that raise of the clock has now come to an end. We must either parallelize all our programs, or face stagnation. There is, however, one avenue for improving performance that is still open: the use of specialized hardware to support the most performance sensitive parts of an otherwise sequential program. Reconfigurable hardware is extremely flexible in this regard, and can produce incredible speedups at a very low clock speed [4,5,6,7]. eMIPS [8] is a dynamically extensible processor that includes a standard MIPS trusted ISA tightly connected to reconfigurable hardware. The programmable logic is divided in *extension slots* that plug into the main pipeline stages during the execution of a program, as depicted in Figure 1. In addition to providing hardware acceleration for improved performance, the extension slots have been used for a variety of other purposes [9,10,11]. In this paper, we describe a scheduling algorithm for allocating the extension slots to competing applications, under a general-purpose operating system. The algorithm and the required hardware support are fairly generic. They apply both to the new, tightly coupled architecture advocated by the eMIPS project and to the more traditional, loosely coupled architectures that use a bus to connect the CPU and the reconfigurable hardware units. It is also possible to consider a GPU used for general computing as a form of accelerator and schedule it according to our algorithm.

Our contributions are:

- A new scheduler that is typically within 20% of the offline optimal schedule.
- The first working implementation on a general-purpose operating system, on real hardware.
- A practical demonstration that scheduling of software threads and hardware accelerators can be realized independently.
- An open platform for experimentation with the acceleration of general application software.

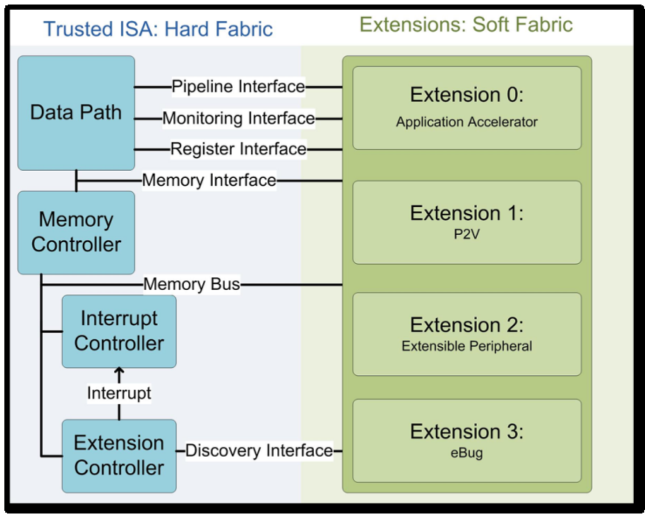

Figure 1: The scheduler supports micro-processor architecture tightly-coupled with a number of hardware *extension slots* usable for accelerating software applications. Accelerators based on more loosely-coupled architectures and GPGPUs can also be supported.

The rest of the paper is organized as follows. Section 2 describes related work in the field. Section 3 describes the hardware support we assume from the underlying architecture. Section 4 describes the new scheduling algorithm. Section 5 presents the software support for the NetBSD operating system that we built around the scheduler, including some basic tools and the additional system services offered by the operating system. Section 6 presents our evaluation of the scheduler, comparing it to other competitive schedulers using synthetic benchmarks and application programs. Results are presented for the Giano system simulator and for three actual systems: the ML40x evaluation board for the Virtex V4 FPGA chip [1], the Virtex V5 based BEE3 [3] and XUP [32] systems. Section 7 lists the detailed set of changes we applied to the NetBSD 4.0.1 source tree to realize our prototype implementation. Section 8 describes the practical procedures for using the resulting system, and Section 9 presents our conclusions.

2 Related Work

Application specific hardware optimizations are not a new idea. The goal is to modify the processor or system platform to allow the application to run more efficiently than on the base configuration. There are two major classes of these optimizations, online and offline. The offline approaches range from static analysis of the application, trace driven analysis, to simulation based design-space exploration [16, 17, 18, 19, 20, 24, 25, 12]. These approaches determine the best configuration for the applications, which is the configuration that will be used.

The online approaches reconfigure the system while the application is executing on the platform. This allows the system to adapt to the application that is being seen, unlike the offline approaches that need to use benchmarks or previous traces to determine the types of applications that would be seen. Most online algorithms fall into one of two types: heuristic, or k-competitive. Heuristics are often good at solving the problem but they usually provide no guarantee on their performance against the optimal configuration. The k-competitive algorithms do provide such a limit. K-competitive algorithms use the optimal sequence of configurations as a comparison point. This analysis proves that even in the worst case the algorithm will be no worse than a factor of k from the optimal result [15]. While a heuristic in the common case may be able to perform better than a k-competitive algorithm, under a worst case input the maximum performance loss is provable for the k-competitive algorithm. For example, Borodin [13] proves that the taskmaster can generate a worst-case time of 2n-1 for the online-optimal strategy by making a move the best option at each step. But Borodin [13] also notes that a scheduler that randomly picks its move (ignoring costs) has an expected execution time that is twice as good on this same sequence.

There are two main types of k-competitive algorithms, the Metrical Task System (MTS) [13] and the k-server problem [14]. MTS is defined in terms of a set of configurations and the cost of switching from one configuration to another. Each application running also takes a varying amount of time to run on each configuration. The goal is to minimize the total time to run a series of applications. The k-server problem has a set of resources that needs to be serviced by the servers. The servers move between the resources using a cost function. The goal is to minimize the total cost to service the requests.

There are different versions of these basic problems. The k-server with excursions is one such variation on the k-server problem, which was first introduced by Manasse [14]. This expansion of the k-server problem has the k servers that are used to handle the requests as in the original k-server problem. However, in addition to moving a server to the resource an excursion can be made from one of the servers. This means that the server will not move but will temporally handle the request. The cost of the excursion is more than if a local server handled the request but less than the cost of moving a server and handling the request. This allows requests to be handled without constant motion of the servers.

There has been much work in using hardware accelerators to improve the performance of applications, see [21] for a start. In many of these works the system mandates that the hardware accelerator is present in the FPGA prior to running the application. With this type of problem there is only one option; the FPGA accelerator must be loaded in order to execute the application. The only scheduling decision therefore is of a batch nature, e.g. deciding in which order to execute (serially) the applications that require the accelerator [22], taking into account the time to load the accelerators [23].

In the eMIPS system [8] hardware acceleration is instead optional, because an application can execute with or without the accelerator enabled. An example code sequence is shown in Figure 2. If the accelerator is not available the application will automatically fall back to the original, un-accelerated software version. It is also possible to transparently disable the accelerator at any time during execution, and to switch the slot to a different application. The accelerator is functionally equivalent to the sequence of instructions inside the Basic Block, and it is triggered by an extended opcode, e.g. the otherwise illegal instruction *ext1* in Figure 2. A load/store memory access inside the accelerator can cause a trap, e.g. a miss in the software-loaded MIPS TLB. In this case the accelerator terminates early and indicates that execution should resume at the corresponding software instruction, inside the Basic Block. Accelerators are automatically generated by the M2V compiler [27].

```
ext1   a0,t1,40     Extension
                    Instruction
sll    t1,t1,1      Basic Block
srl    t3,t0,31     Instructions
or     t1,t1,t3
sll    t0,t0,1
srl    t3,a0,31
or     t0,t0,t3
sll    a0,a0,1
srl    t3,a1,31
or     a0,a0,t3
sltu   t3,t1,a2
beq    zero,t3,40
sll    a1,a1,1
```

Figure 2: *Extension Instructions* trigger the activation of hardware accelerators in eMIPS. If the accelerator is not enabled the extended opcode *ext1* is treated as a No-op and execution continues through the rest of the Basic Block. Otherwise the accelerator executes and the Basic Block is skipped. An interrupt or trap inside the accelerator can cause execution to resume somewhere inside the Basic Block.

The eMIPS system has a capacity problem, because the number of accelerators that can be loaded at any one time is limited. It presents a performance optimization problem, because an application will execute more slowly without the accelerator.

In this type of system the scheduling algorithm performs two practical choices during execution; select the best (set of) accelerators to enable, and select which accelerator to remove so that a new accelerator can be loaded onto the FPGA.

3 Hardware Support

What any accelerator scheduling algorithm really wants of hardware is a reliable measure of the costs and benefits provided by the accelerator resources. The cost is the cost of *not* enabling an accelerator for the application that demands it, and the benefit is the performance advantage generated by an enabled accelerator. Note that some algorithms can use just the first measure.

eMIPS provides the desired cost/benefit measures as a set of counters, similar to the miss/hit counters of a hardware instruction cache. Figure 3 illustrates the structure of the combined software-hardware system. The miss counters are global, and are depicted by the blue boxes tagged *Misses* in the bottom-half of Figure 3. There is one counter per (selectable) extended opcode. A miss counter increments when its corresponding extended opcode is *not* recognized by any of the loaded accelerators. The hit counters are instead local to the extension slots, and are tagged as *Hits=n* in the green boxes that represent the extension slots in the bottom-half of Figure 3. There is one hit counter for each extension slot. The hit counter is incremented every time the accelerator recognizes and executes an Extension Instruction. Basically, this is the count of how many times the corresponding accelerator has been activated.

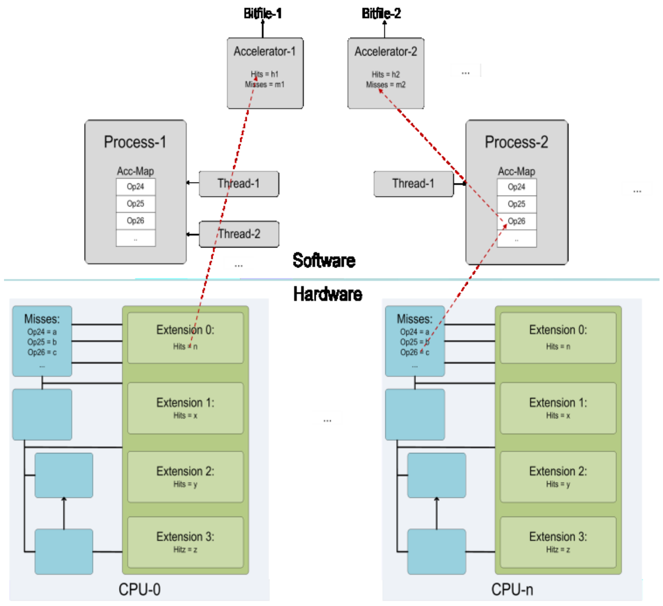

Figure 3: Hardware support for scheduling is based on *hit* and *miss* counters. Each time an Extension slot recognizes an Extension Instruction the corresponding hit counter is incremented. Each time an extended opcode is not recognized by any of the slots the corresponding miss counter is incremented. The mapping from extended opcodes to accelerators is provided by software.

Note that while the hit counters directly refer to an accelerator, the miss counters do not. It is up to software to maintain the mapping between the extended opcodes and the accelerator that is supposed to recognize a given opcode for the currently executing thread. This is illustrated in Figure 3 by the dashed red arrows linking counters to accelerators. The arrow that links a miss counter with an accelerator jumps through a mapping table, whereas the hit counters are directly linked. It is also up to software to decide what abstraction to associate with the accelerator-opcode mapping table (*Acc-Map* in the top portion of Figure 3).

An important factor in scheduling is the cost of loading the accelerators themselves, see [23] for a recent discussion. The implementation on eMIPS uses the smart DMA engine described in [30], wrapped in a simple, asynchronous device driver interface. Computation therefore continues while the accelerator is being loaded. Experiments [30] indicate that the loading time is independent of the actual content of the configuration bitfile. It only depends on the length of the file, divided by the bandwidth between the DDR memory and the ICAP port. Since the 32-bit ICAP port has a maximum clock of 100 MHz the bandwidth attainable is 400 MBps (unless overclocked). In reality, there is hard competition for the DDR memory interface and the results are much lower. In addition to the Ethernet interface, the current eMIPS processor does not have any caches. The processor therefore uses a lot of DDR bandwidth, and disrupts the (four) open-bank caching performed at the DDR chip itself. Under the NetBSD OS, large transfers are typically split into single-page transfers of 4KB and on the XUP board we measure an effective user-mode bandwidth of only about 800 KBps. With a stand-alone test program we can reach 225 MBps, which closer to a kernel-mode transfer bandwidth. This value was confirmed by hardware measurements made using ChipScope.

Regardless of the actual bandwidth, for scheduling purposes the key fact is that the file length is a valid measure of the accelerator loading cost. In practice, the hardware synthesis tools tend to fill up all the available area for an accelerator slot, which leads to a file size that is constant among all accelerators. The value does depend on the slot area size, which is a hardware design time parameter. On actual eMIPS systems the file sizes hover around 100 KB for the Virtex V5 systems and 120 KB for the Virtex V4 systems. The minimum theoretical accelerator load time therefore is 250 microseconds. The minimum time on an XUP board is 500 microseconds.

3.1 Discussion

Without any performance indications from hardware, scheduling of a set of accelerators is very simple. Software has no basis to make any scheduling decision and therefore it can only allocate the accelerators in a first-come, first-served manner. Software could use the completion times of certain programs from past executions, but this information can be input-dependent and unreliable. Involving humans in the decision making is insecure and unacceptable in a multi-user environment. Software can be fair in allocating slots to multiple users, but the majority of general purpose computers today are used by a single user. A similar problem arises with pre-emption. A few long-running programs can hog the accelerator slots and never allow any other program to use the accelerators. On what basis would the scheduler choose which application to pre-empt? Not knowing whether the accelerator was useful or not, the most sensible choice is the obvious: round-robin, perhaps with priorities. It is hardware support that therefore makes a difference, and opens up the field to a number of possible optimizations. For example, the MTS problem formulation assumes knowledge of the scheduling costs.

We developed a scheduling algorithm that is independent of the thread scheduler. The concept is that CPU and accelerator slots are two different physical resources and should be scheduled independently and orthogonally. It is an interesting question whether the two resources are indeed, as they appear, truly independent of each other. The argument in favor is that acceleration only affects the completion time for a thread's computation; it does not affect its behavior. The argument against is that altering the order of events can make the thread scheduler alter its decisions, therefore they are not independent. Our assumption is similar to the assumption of independence between processes made on general purpose operating systems. It is *not true* that a process fully virtualizes a system and therefore does not affect any other process. Classical counter-examples are priority inversions, communication via pipes, gang scheduling, and many others. Nonetheless, the assumption in practice works and it is therefore the norm.

When we say "cost" and "benefit" we make a hidden assumption about the "optimality" criterion. What we really mean is that we intend to minimize the completion time for a set of tasks; this is our only measure of success. We (e.g. the scheduler of a general purpose OS) do weight this goal for "fairness" among users, but we do not consider other important elements such as responsiveness [33] and assume that a simple tweaking of priorities can solve the problem. The optimality criteria in an embedded system are even more complex and involve guaranteed completion times, power, physical dimensions, monetary cost, reliability and more. We could argue that an accelerator simply reduces the task computation time and therefore the system will still meet the deadlines without changes. But it is easy to build a counter-example using priority-inversion in a (poorly programmed) system.

We considered a few alternatives that we can realize on eMIPS. One idea is to measure the cycles actually spent in the accelerator. The hard fabric can securely and reliably take this measure starting a counter from the moment it enables the accelerator for one opcode to the time it disables the interface signals. This measures the benefits; it is more difficult to measure the costs. Perhaps on a miss we can start a cycle counter and stop it on the first branch… but what of multi-branched software. Without a corresponding reliable measure of the costs, the time spent inside the accelerator is not in fact a real benefits measure. One advantage of this approach is that it provides concrete performance feedback to the user. One disadvantage is that it links too tightly the counters to one specific execution model. What about accelerators running in parallel to software, or larger accelerators that involve many software branches? There is also potential for abuse, for instance an accelerator that occasionally holds the interface for a very long time, only to bump up its perceived benefit measure. We decided to provide a more generic meaning to the counters themselves, and leave this and other valid alternatives for future work. Note that the scheduling algorithm itself is independent of the actual unit of measure for "cost" and "benefit", no changes are required if the hardware changes.

In our implementation using NetBSD we placed the accelerator-opcode mapping table in the process data structure, logically equating the accelerator resource to a section of executable code. Other choices are possible, depending on the specifics of the selected operating system. On one extreme the mapping table can be a global resource, for instance in the RTOS of an embedded system. On the other extreme the table can be per-thread. Mapping structures other than arrays are possible.

4 Scheduling Algorithm

The pseudo-code for the scheduling algorithm is shown in Figure 4. Each accelerator has an associated *activity history*, which in the eMIPS case is the weighted sum of hits and misses. Generally speaking, the algorithm uses the past-predicts-future paradigm, and attempts to reach a (stable) state where the new level of activity for each accelerator is the same as in the past. The more the future deviates from the past, the faster we drop the relevance of the activity history. The level of activity for each accelerator defines, according to its own expected benefit, the accelerator's *current utility*.

The algorithm makes two passes over the list of accelerators. The first pass simply computes the overall decay factor. Note that only accelerators with some level of activity contribute to this first computation. The second pass re-computes the current utility for all accelerators, whether they are loaded or not. At the end of the second pass we know which one is the most useful, not-loaded accelerator *N*, and which one is the least useful, loaded accelerator *L*. The scheduler loads accelerator *N* if it is more useful than *L*, or if there is a free slot available. The scheduler will load at maximum one accelerator per scheduling round.

Note that the algorithm does not pro-actively remove a loaded accelerator, even if no application uses it. This means that during idle time no changes are performed and the system can quickly restart without penalties. This lazy strategy also applies to "empty" slots, e.g. slots that have an accelerator loaded in them but no corresponding active application. Should the user re-invoke a just-terminated application the system may realize that the corresponding accelerator is still loaded and re-activate it at zero cost. This optimization implements the LRU-with-second-chance replacement policy [28, 29] for the loaded accelerators. Note, however, that a very small number of slots will make this optimization moot in many cases.

```
Schedule::
  Delta = $\sum_i$ (OldActive$_i$ - NewActive$_i$) / NewActive$_i$
  Decay = Delta / AcceleratorCount;
  Forall (i)
    OldActive$_i$ = ((Decay * OldActive$_i$) + NewActive$_i$)/2
    Utility$_i$ = (OldActive$_i$ * Benefit$_{i)}$ - LoadingCost$_i$
  BestNotLoaded = Max(Utility$_i$)
  WorstLoaded = Min(Utility$_i$)
  If (BestNotLoaded ≠ NULL)
    If  WeHaveAFreeSlot(&s)
      Load(BestNotLoaded,s)
    Elif  BestNotLoaded.Utility > WorstLoaded.Utility
      Load(BestNotLoade, SlotOf(WorstLoaded))
```

Figure 4: Pseudo code for the scheduling function. At each invocation, the new amount of activity *NewActive* for each accelerator is compared against the accumulated, predictive history value in *OldActive*. The deviation *Delta* defines the *Decay* ratio for the previous history. The new activity is then added into the decayed history, and used to compute the new *Utility* of each accelerator. The algorithm loads a new accelerator if it is more useful than at least one of the loaded accelerators, or if there is a free slot available. The *LoadingCost* of an accelerator varies; in practice it is proportional to the file-length of the accelerator.

Only a minor detail has been omitted from the pseudo-code of Figure 4: the option to *lock* an accelerator and prevent the scheduler from removing it. This option is not normally used; it was introduced to support optional higher-level policies and/or user preferences. Locked accelerators are ignored in the selection of the least useful accelerator.

5 Software Support

The scheduler is likely an integral part of the operating system kernel and only requires simple monitoring tools, to help system administration. Even our implementation as an optional, loadable kernel module exploits only existing facilities and no new tools are required for loading or unloading the scheduler. In addition to its own implementation (see Figure 4) the scheduler requires some machine-dependent code to access the hit/miss counters. Figure 7 shows this code for the eMIPS case.

A simple way to aid monitoring is a system call that returns the list of accelerators, and all the information the scheduler maintains about them. A set of flags indicates if an accelerator is for kernel (privileged) use, private or shareable by all applications, whether it is currently loaded and/or locked, a count of active applications making use of it, the hit and miss counters, and the benefit estimates. Utilities similar to top(1) and TaskManager can then help visualize this information for the user.

The scheduler does not use priorities, therefore there is no need for tools to manually tweak them, such nice(1) or TaskManager. Note, however, that eMIPS accelerators depend heavily on the behavior of their software counterparts, therefore altering the priorities of the applications themselves has a direct corresponding effect on the accelerators they use. On the other hand, the locking functionality does require a simple manual tool, and potentially can lead to a more sophisticated, long-term, online monitoring and optimization infrastructure.

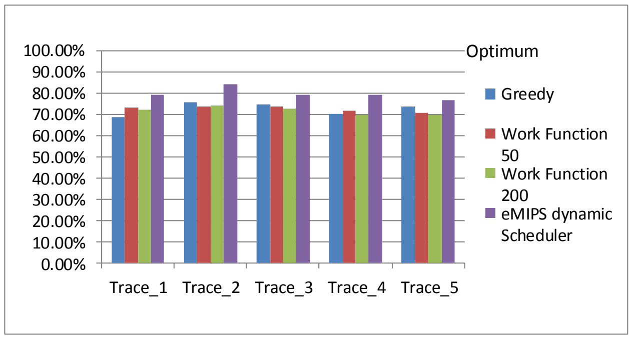

Figure 5: Comparison of the new scheduler with other known schedulers, using synthetic traces. In all cases the new scheduler gets closer to the ideal optimum, being on average 10% better.

In the implementation phase we spent more time and code on a completely separate issue: the specifics of loading the accelerators themselves. An accelerator in the eMIPS case is a file with a new, vendor-specific structure. It contains the bit-stream data to be fed into the internal re-configuration port for the FPGA. In addition to recognizing the file format, we need to be able to associate the accelerator with one or more applications. This was done by defining a new file format, the Secure Executable (SE) format described in [31], essentially the (backward compatible) concatenation of an ELF image file with the accelerator bitstream, plus a header and a security digest. Support for the new file format includes a simple utility for creating an SE image, a utility for displaying the content of an SE image, and modifications to the system image loaders to support the SE format. The loader code had to be replicated once in the kernel proper, and once for the shared library loader. Debugging the shared library version was not trivial.

There are additional tools in the eMIPS systems for profiling applications and for automatically creating accelerators, but their description is outside the scope of this paper.

6 Evaluation

We performed two sets of evaluations: in a controlled simulation environment, and on the actual systems. Given the nature of some algorithms, we could only make a meaningful comparison between algorithms in simulation, using as input a common set of traces. Section 6.1 describes this first set of experiments. However, simulation does not always capture the complexity of a real system and therefore we performed a second set of experiments to test the eMIPS scheduler on the various systems where eMIPS runs. This includes the Giano full system simulator, and various Xilinx-based systems. Section 6.2 describes this second set of experiments.

6.1 Simulation

The program used for simulation is shown in Appendix A. In one mode of invocation, the program generates an activation trace according to some desired features. In a second mode of invocation, the program uses a trace file as input into the desired algorithm and produces the estimated completion time.

The algorithms we used in the comparison can be summarized as follows:

- *Optimum*: This not an algorithm but a value, obtained assuming that there is an infinite number of slots and all accelerators are always loaded, at zero cost. This ideal value provides the maximum speedup attainable for the given set of applications.
- *Greedy*: At each round, this algorithm selects the accelerators with the highest expected benefit and loads them in the empty slots. If there are no more empty slots nothing is done. Accelerators are unloaded upon task termination. This is a similar algorithm as defined in Section 4.2 of [23] but, like most of the other algorithms we compare, it does take into account the penalty for loading a new accelerator during the benefit computation.
- *Work Function 50*: This is the algorithm defined in Section 4.3 of [23]. It uses dynamic programming and is therefore too expensive to use in many online settings. To make it online, the offline optimal algorithm is applied to the current task plus the previous fifty tasks in the past history.

- *Work Function 200*: Same as above, but with a longer history buffer.
- *eMIPS*: Our algorithm, as defined in Section 4.
- *Basic 0.7*: Similar to our algorithm, but with a fixed decay factor.

The Optimum sets the 100% bar for performance, and we report the results for the other algorithms relative to this ideal maximum. The Greedy algorithm is perhaps the simplest, one-step look-ahead strategy that still takes into account all the elements of the problem. We would expect this strategy to set the lower bar to compare all the others to. The WorkFunction algorithm is an online adaptation of the offline optimal strategy. We would expect that the offline algorithm would degrade gracefully with the reduced information available online and to make this strategy competitive. Finally, the Basic algorithm is similar to the eMIPS one, but discards past history at a fixed rate. This is slightly more efficient to implement, but we would expect this algorithm to be less reactive to e.g. phase changes and therefore perform similarly but sometimes worse than eMIPS. The simulator does not take into account the cost of running the algorithms themselves. They are all assumed to take zero time. Algorithms are run every 13 million cycles.

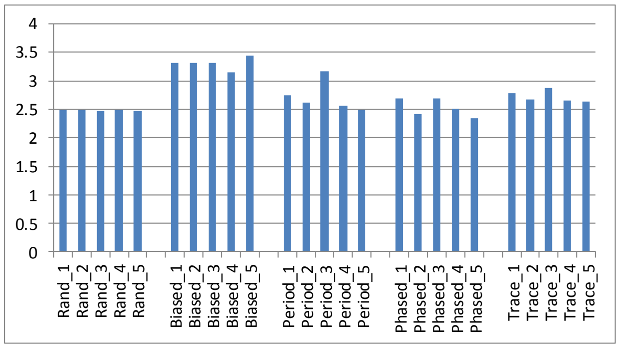

Figure 6: Ideal speedups for the synthetic traces, obtainable only with a large number of accelerator slots, and zero accelerator load times. The eMIPS scheduler reaches about 80% of the ideal speedups, on average.

A trace file is a sequence of pairs (*ID,Cycles*). Each pair indicates that the application uniquely identified by the integer *ID* became schedulable for the given number of *Cycles*. All traces use a maximum of seven different applications, with expected acceleration factors of {3.96, 2.59, 3.17, 3.06, 1.12, 3.75, 2.64} with a geometric mean of 2.72 and arithmetic mean of 2.9. The simulated system has four accelerator slots. The durations are all randomly selected, with a uniform distribution between 10 and 30,000 cycles. Traces are all 100,000 pairs long.

The traces we used in the comparison can be summarized as follows:

- *Rand_n*: These traces are generated with a different random seed each, and with uniformly randomized application selection. This is the standard, white-noise type of un-biased input.
- *Biased_n*: In these traces the application selection is biased towards a certain set. Different traces have a more remarked bias towards a smaller number of applications. Here we are simulating the small workload typically generated by a single user, or on some dedicated server.
- *Period_n*: These traces are for a set of applications with periodic activation times: the same applications repeat in the same order within each period. The traces have periods of length 15, 15, 10, 10, and the whole trace. This is the case of a soft-real-time system, or a machine dedicated to a repetitive taskset.
- *Phased_n*: These traces are equally subdivided in ten phases. In each phase only a given subset of 3-6 applications is active, though their activation order is random. This is the case of a set of programs that change their behavior while they are performing their task.
- *Trace_n*: These traces are a concatenation of the (corresponding) previous four traces. They summarize the average performance of an algorithm.

Given the practical considerations of Section 3, we assume that loading an accelerator in all cases has a fixed cost of 112,000 cycles. The simulator assumes that the loading process stops execution. This is incorrect, we are using DMA and the processor instead continues. But as noted, the bus utilization will slow down the processor by a certain amount (unknown). To account for both factors, we use a fix load time much lower than the 250 us best.

A summary of the results is shown in Figure 5, using just the combined traces. The complete set of results is shown in Figure 8. All algorithms realize a sizeable percentage of the optimum, ranging from a worst case of 69% for *Greedy* in *Trace_1*, to a best case of 84% for *eMIPS* in *Trace_2*. This suggests that taking into account the expected speedups in the scheduler is sufficient to reach some 70% of optimal, the specific strategy being only a secondary element. Looking at the details however, we notice a wider spread: the absolute worst result is a 58% for both *Work Function* in *Period_3*, and the best is a 98% for *Greedy* in *Biased_1*, *2*, and *3*.

Contrary to expectations, the *Greedy* algorithm is the worst overall performer in only one out of the five combined traces; in *Trace_2, Trace_3, and Trace_5* it outperforms both of the *Work Function*, although not by much. Looking at the detailed results, we see that *Greedy* is actually the overall winner in 15 out of 25 cases. *Greedy* always wins in the *Rand* traces with a consistent 67% of optimal. *Greedy* also wins in the *Biased* traces, where it reaches the three best results overall, with 98% of optimal. And again, *Greedy* is the winner in the *Period* traces. On the other hand, in the *Phased* traces *Greedy* is almost always the worst performer, by a 12% to 20% range.

The Work Function algorithms do not perform as expected. Not only are they the worst overall, but they also show inconsistent results. They performance is similar, but the shorter history wins by the largest margins (about 4%) in *Period_5*, *Phased_1*, and *Phased _4*. The long history wins at best by 1.7% in *Rand_4*. Note that in the *Period* traces both histories are long enough to cover the base period. Clearly the offline *Work Function* algorithm does not quite scale as hoped for when made online.

The *eMIPS* algorithm overall performs better than the others, by about 10% on average. Looking at the details, however, we note that *Basic* comes very close to *eMIPS*, and it is in fact slightly better in the *Rand* traces, ranging by 0.2% to 0.3% better. As expected, *eMIPS* does get the advantage over *Basic* in the *Phased* traces, by 3% to 5%.

The comparison between Greedy and eMIPS is the most interesting. When Greedy wins, it is never by a large factor. The range is 1.0% to 2.7%, with an average of 1.5%. When eMIPS wins on the other hand, the difference is much more visible: 3.0% to 19.3%, and 10.1% average.

The optimum speedups are shown in Figure 6, with each trace resulting in a unique speedup. Speedups vary between 2.3 and 3.4, with an average of 2.75. *Biased_n* traces show a higher average speedup, due to the bias towards applications with smaller IDs that have higher acceleration factors. The other cases are closer to the average, except for the 3.2 of *Period_3*. In this case the application mix has an average speedup of 3.25, higher than usual, and very close to the result obtained over the whole trace. Since the durations are randomly chosen, this is to be expected.

6.2 Actual Systems

To evaluate the eMIPS scheduler we perform a set of experiments on the Xilinx XUP board and on the Giano simulation of the same XUP board. In addition to providing insights on the practical behavior of the system, this process validates simulation. It does not compare the various algorithms.

6.2.1 The TLOOP Probe

The program shown in Appendix B can simulate various application behaviors by tuning three parameters: the amount of work that can be accelerated and therefore performed in the accelerator (*count*), the number of times the accelerator is invoked (*nloop*), and the amount of work that cannot be accelerated (*nwork*). If we link the TLOOP image with a hardware accelerator (SE image) we will provide an input to the scheduler. If instead the image does not contain an accelerator (ELF image) the program will always execute in software, unaccelerated. The scheduler implementation effectively identifies an accelerator with the file that contains it. Therefore we can make copies of the SE images to fool the scheduler into thinking it is working with many different accelerators... even though they are all identical.

Note that our evaluation targets the scheduler itself, not any particular application or interesting usage scenario. Using the TLOOP probe reduces the amount of noise in the experiments, and therefore aids our understanding of the behavior of a complex system. Evaluation of how the eMIPS system performs on real applications is outside our scope.

The probe's wall-clock execution time is a function of the three input parameters:

$$t = tloop(count, nloop, nwork) =$$
$$C + nloop * (L + A * count + W * nwork) \quad (1)$$

where C is the time to activate the process, L is the per-iteration cost of the first loop in the *main()* function, A is the per-iteration cost of the assembly loop, and W is the per-iteration cost of the second loop. We can optionally use the timing report from the program itself to eliminate the factor C.

The execution time of the accelerated version, using the Verilog code of Appendix B, effectively eliminates the effects of *count*, and the function reduces to:

$$t = tloopA(nloop, nwork) =$$
$$D + nloop * (B + W * nwork) \quad (2)$$

where D now includes the accelerator loading time, and B is very close (but not identical) to $L+A$. Note that equation 1 has the same linear shape of equation 2; using a nested loop just gives us more fine grain control on the second term. Table 1 summarizes the values we measured for the constants in equations 1 and 2 on the Xilinx XUP board.

Table 1: TLOOP execution time depends on the input arguments and on the process and accelerator load times.

Dependency	*Time(ms)*
C: process load	1310
L: nloop	0.011620
A: count	0.001864
W: nwork	0.006414
D: accel. load	**TBM**
B: nloop	**TBM**

The accelerated version of the probe (TLOOP_A) uses the Secure Executable image format [31] which is slightly more expensive to load (see Section 5). The load cost *D* of equation 2 includes the cost of loading a regular ELF image (e.g. *C* from Table 1), plus two additional costs as shown in equation 3.

$$D = C + S + V + I \qquad (3)$$

The first cost *S* is the check to see if the image to load is in the SE format or not. This cost is incurred on all images; it is small but it could be eliminated by breaking the ELF compatibility, e.g. using a bit in the ELF header. The second cost is only incurred on SE images, and again has two parts. The first is the cost *V* of verifying the integrity of the file itself, which at minimum involves a CRC over the whole file. Stronger security digests are typically more expensive. The second is the cost *I* of loading the accelerator bit file into the ICAP programming interface of the FPGA. Note that *S* is constant, but *V* and *I* scale linearly with the length of the file. The values we measured on the XUP board are reported in Table 2.

Table 2: Breakdown of the image loading time for the accelerated probe version.

Constant	Time (ms)
S	60
V	190
I	0.498
D	TBM

A number of parameters depend on the size of the corresponding object. Table 3 shows the sizes for the two versions of the probe. The software portion (text+data) is the same for both versions, and much smaller than the hardware portion.

Table 3: Sizes in bytes of the scheduler and probes.

File	Text	Data	Accel.	FileSize
TLOOP	3,500	228	0	8,095
TLOOP_A	3,500	228	109,604	117,812
Scheduler	7,832	80	0	15,508

In principle, we can measure all the constants in the two equations using just the values (0,0,0), (0,1,0), (1,1,0), and (0,0,1) for the input triple (*count, nloop, nwork*). In reality, there are a few caveats to bear in mind while using the probe: The absolute values in the set {*L,A,B,W*} are much smaller than those in the set {*C,D*}; there are elements of variability in the load times; and the time reported by the *time(1)* cover program has a limited granularity of about 30 milliseconds. Very small input triples therefore will generate time costs that are either overwhelmed by the load times, not measurable with the *time(1)* facility, or both. TLOOP's internal time reporting, using *gettimeofday(2)* (GTOD), is instead much more precise. Using the board's 10MHz clock this facility has a granularity of 1 microsecond.

Other potential sources of large errors in the measurements include network traffic and paging. To avoid them, the Ethernet cable should be disconnected from the board if possible, and the system should be idle, preferably in single-user mode. All programs should be run first to page them in from disk. Note that to measure the scheduler we are interested in somewhat long running applications, therefore the inputs to TLOOP will be fairly large integers. This eliminates the noise due to granularity issues. The lack of caches on the XUP implementation is a determining factor for TLOOP's predictability.

Linearity

This section evaluates TLOOP as a function of the individual parameters. The upshot is that the program is perfectly linear for large values and when measured by *GTOD*, and shows some non-linearity only for very small input arguments if measured with *time(1)*. The accelerated version of the program incurs some additional fixed costs.

Table 4: TLOOP single argument fits $t = a_0 + a_1 * x$.

Input	Measure	a0	a1	R2	MinVal
nloop	time(1)	1310	0.011620	0.9964	10,000
	GTOD	0.431	0.011620	1	n/a
count	time(1)	1310	0.001864	0.9649	80,000
	GTOD	0.443	0.001864	1	n/a
nwork	time(1)	1310	0.006414	0.9816	100,000
	GTOD	0.431	0.006414	1	n/a
nloop_A	time(1)	1560	0.007262	0.9902	100,000
	GTOD	0.430	0.007262	1	n/a

A linear equation $t = a_0 + a_1 * x$ matches all measurements, as shown in Table 4 for various combinations of measuring facility, input parameter, R^2 fit for small values, and start of the linear range.

Figure 9 show the general behavior of TLOOP over a large range of inputs. In this case we used *nloop* and measured using *time(1)*. The graphs for the other input arguments show identical trends, and so do the measures we obtained using *GTOD*.

Figure 10 shows the non-linearity at small input values for TLOOP against *nloop*, when measured with *time(1)*. The probe is linear only for values larger than approximately 10,000. Figure 11 uses *GTOD* for the same measurements and is instead perfectly linear. Similar results apply to the *nwork* parameter, shown in Figure 12 to be linear above 100,000 and to *count,* shown in Figure 13 to be linear above 80,000.

The row *nloop_A* in Table 4 reports the results for the TLOOP_A accelerated version of the probe. There is no dependency on *count* because the accelerator returns immediately (see Eq. 2). The dependency on *nwork* remains unchanged. The dependency on *nloop* changes because the function *loop()* now consists only of one (extended) instruction instead of 5 instructions as shown in Appendix B. Figure 14 shows the non-linearity with *time(1)* for values less than 100,000.

6.2.2 Scheduling tests

The current build of eMIPS for the XUP board only provides one accelerator slot, with four slots projected in the near future. Consequently, some interesting test scenarios could only be run on the Giano full-system simulator. The tests that could be run on both platforms validate the fidelity of the results on Giano.

The most visible functionality of the scheduler is to assign the (one) available slot to the most valuable program. We can test this with our probe using two accelerated copies, one that uses the accelerator frequently and one that uses it infrequently. Changing the *nwork* parameter can accomplish this:

- Infrequent is INF="tloop_a 10000 1000 10000"
- Frequent is FRE="tloop_b 10000 1000 1000"

where *tloop_b* is a copy of the accelerated probe. We can use the shell's job control facilities to try various scenarios, manually and using scripts. Table 5 reports the results of the experiments. In the parallel execution FRE||INF of the two probes we start FRE first; in INF||FRE we start INF first. The rows marked (FRE) indicate the elapsed time at which the first job completes. All times are fairly consistent across runs; variations are below the 0.1 second mark.

Table 5: Two probes competing for one slot. One uses the accelerator INF-requently, the other more FRE-quently.

Job	XUP/a	XUP/u	Giano/a	Giano/u
INF	63.9	76.0	59.2	74.8
FRE	7.9	20.0	8.5	22.0
FRE||INF	76.9	99.9	71.3	101.5
(FRE)	15.0	39.7	14.3	43.8
INF||FRE	76.7	99.9	71.6	96.0
(FRE)	15.0	39.3	14.8	43.8

When run individually, both jobs complete faster with acceleration ("a" columns) than without ("u" columns). The two jobs get a different speedup from acceleration: about 1.2x for INF and 2.8x for FRE.

The times on the XUP board are mostly higher, and between 1% and 10% of the results on Giano. The longer running jobs are more precise, between 2% and 7%.

The order in which we start the two jobs in our script does not matter; the FRE||INF and INF||FRE rows show similar results. For both starting orders the shorter job completes first, as expected. The completion times for (FRE) are twice the time for the isolated, accelerated execution of FRE. This is expected under the assumption that the CPU is allocated fairly. The columns without acceleration demonstrate the fairness of the thread scheduler, who allocates 50% of the CPU to both competing programs. The columns with acceleration also show that the CPU was allocated 50% each. The accelerator scheduler allocated the one slot sequentially; first to the FRE job and when this completes, to the INF job. Therefore the elapsed times are the sum of 2*FREa (jobs running in parallel) plus the remainder of INFa. The first portion is what is reported as (FRE).

Table 6: All combinations of running two probes with one slot. Probes run accelerated and unaccelerated, sequentially and in parallel, on the XUP board and on Giano. Some commutative entries are omitted.

	XUP elapsed	XUP GTOD	Giano elapsed	Giano GTOD
INFu+FREu	**99.8(75.9)**	**74.7+18.7**	**101.4(74.5)**	**73.5+22.0**
INFa+FREa	**77.1(63.9)**	**62.6+ 7.2**	**70.9(58.5)**	**57.3+ 7.3**
INFu+FREa	**87.8(75.8)**	**74.6+ 6.4**	**85.4(74.5)**	**73.5+ 5.9**
INFa+FREu	**87.7(63.9)**	**62.6+18.6**	**85.8(58.9)**	**57.7+22.1**
INFu||FREu	**99.7(39.4)**	**93.6+37.4**	**101.6(46.4)**	**96.0+44.4**
INFa||FREa	**77.9(16.3)**	**70.2+12.5**	**71.7(14.3)**	**65.0+12.0**
INFu||FREa	**87.7(15.3)**	**81.4+12.9**	**85.8(14.2)**	**80.0+12.0**
INFa||FREu	**87.6(39.9)**	**81.2+37.6**	**85.0(45.6)**	**79.0+43.5**

Table 6 shows the results from running all combinations of the two probes, running sequentially or in parallel, accelerated and not. Some entries are omitted because sequential execution is commutative, and the starting order of a parallel job is not relevant. The "elapsed" columns are computed as in Table 5, using *time(1)*. The GTOD columns report the elapsed time as observed from within the probe, eliminating scripting and startup costs. Each member of a timing pair refers to the corresponding probe, indicated in the first column. The sequential execution of the accelerated probes gives the best result. A close second is the parallel execution of the accelerated probes.

Stability
We use the term "stable" to refer to a desirable property of the scheduler, namely that it should not constantly reallocate the resource among competing clients. Even though loading an accelerator is not a huge overhead, still we would like the scheduler to demonstrate a certain degree of stubbornness once it reaches a decision to load. We can use two identical jobs to test for stability in the scheduler's decision making. All things being the same, the scheduler should keep the accelerator allocated to the job that was started first. We use two FRE probes in parallel for this test. Unlike the regular thread's scheduler, the accelerator scheduler is invoked infrequently enough that we can insert printouts without triggering insanity. Therefore we simply used debugging printouts to verify that the scheduler made its decision.. and kept it. This test is very simple, and yet it was rather useful during development. We noticed that most implementation errors somehow resulted in one form or other of instability.

Scheduling overheads
In the absence of any other accelerators present, the parallel jobs complete in the same time with and without the scheduler loaded. To test the effect of the presence of accelerators in the system we can start and immediately stop a number of accelerated probe copies, then run the unaccelerated parallel job FRE||INF. Even though all the probes are suspended, the scheduler still must scan them all (to decay their history) and therefore will create some small amount of overhead. We used up to 20 suspended probes and were not able to measure any difference.

Measuring the effect of the accelerator load time is not trivial. As mentioned, the size of an accelerator is effectively constant, and the ICAP interface is much faster than the processor. Therefore to measure any difference we need a very large accelerator, which is impossible to create because they are all the same size. What we can do instead is to load the same accelerator multiple times, e.g. concatenate multiple times the same ICAP file into a larger accelerator image.

<More tests to be added here in final version>

7 Changes to NetBSD

In this section we describe in detail the changes applied to the NetBSD source tree to support the eMIPS processor and the accelerator scheduler.

In December 2008 we started the eMIPS port from the source code base of the then-current official release, version 4.0.1. The basic strategy for supporting eMIPS was to clone the existing machine-dependent code for the PMAX (aka DecStation series of workstations) and apply all the required changes. This entailed a larger number of changes than minimally required, but in the end produced a complete, self-hosting system that we can trust to be relatively bug-free. Modulo the byte-order, any software produced for the PMAX in the past 20 years will work on eMIPS because the eMIPS ISA is a superset of the R2000 processor. Indeed, we found very few software bugs, only one of them serious (in the GCC compiler). The system has now been in use for years and we never observed even a single crash.

We wrote additional code for supporting soft-floats on MIPS, a bus driver for the eMIPS on-chip peripheral bus, device drivers for a number of new peripherals, boot loader code for disks and Ethernet, and miscellaneous other code. One required feature we added to the existing MIPS code is the ability to access I/O space at arbitrary physical addresses, mapping them virtually. Existing code for MIPS assumed I/O space lived entirely in the K1SEG space, e.g. the first 512 MB of physical space. Similar changes allow the system to access 4 GB of memory when available, e.g. on the BEE3 system [3].

Bootstrapping the system required cross-compiling the whole tree and building a bootable disk image. We used a VirtualPC running NetBSD 4.0.1 as a host, and the Giano simulator [2] with an ML40x configuration as the target. Cross-compilation of the whole NetBSD tree on a notebook takes about 15 hours. The same compilation takes about a month when done natively on the simulator. To help the installation process, we added a second SystemACE (disk) peripheral in the simulated ML40x configuration and used it to access the ISO image of the distribution CD we had created. This turned out to be about as quick as accessing the distribution files over the Ethernet, directly from the VirtualPC host. This procedure is more fully described in Section 8. The system was operational in simulation in just a couple of weeks, over the 2008 winter holidays. Another week was then spent providing diskless booting and NFS support.

Once the system was fully operational and tested we applied the changes to support the new accelerator scheduler. To minimize the amount of changes in the OS kernel and provide maximum flexibility we realized the scheduler as a Loadable Kernel Module (LKM), with very simple interfaces to the kernel. All changes are conditional, with absolutely no effect on other architectures. Indeed, the longest piece of additional kernel code is 93 lines in one single function, *accelerator_switch()* which is invoked at thread context switch time. Here we collect the performance indicators used by the scheduler. Machine-dependent code provides access to the hit/miss counters, and the slot control registers. In the machine-independent portion of the NetBSD kernel code the only other additions are conditional function calls into the scheduler at process termination, system-wide process priority re-computation, and image activation.

We added support for the Secure Executable (SE) image format [31], both in the kernel and in the shared library loader. The SE format links a software image and a hardware image (bit file) in a single file, protected against tampering by a security digest. Support for the SE format required more code than the scheduler itself, especially since it is duplicated in user and kernel space. The LKM source file for both scheduler and SE image loader is 893 lines long. It compiles to 6,096 bytes of MIPS code.

When the LKM is not loaded there is no measurable overhead in the system. The small memory cost for the LKM is avoided. All function pointers are NULL and the corresponding indirect calls are avoided. The context switch code recognizes the absence of accelerators with a single test. Therefore the system is fully backward compatible and works as expected in the absence of specialized applications. When the LKM is loaded but there are no accelerators in use the overhead is also below the measurable threshold. The function calls described next are performed, but they return immediately to the caller.

When there are accelerators in use there are additional costs in three places. During priority re-computation (about once a second in *schedcpu()*), the scheduler scans the list of loaded accelerators and re-evaluates their merits (see Figure 4). This can occasionally lead to an accelerator load/unload event, which is expensive. Otherwise the cost is linear with the number of accelerators loaded, because the scheduler makes a single pass over the list (note that this is dependent on the specific scheduler loaded). At context switch time, there is a potential additional cost for gathering the counters and enabling the correct set of accelerators. This cost is only incurred if/when switching to/from a process that actually uses an accelerator and is therefore avoided when no accelerators are in actual use, e.g. if they are quiescent. The incurred cost depends on the number of accelerator slots configured and the number of opcodes supported. The maximums for the eMIPS system are currently four slots and eight opcodes, respectively. The actual number of accelerator slots is dynamically configured at system boot time. Figure 7 shows the pseudo code for the *accelerator_switch()* function, invoked by *mi_switch()*. A simple comparison captures the case when no accelerators are involved in the context switch. If the number of active accelerators is small this test will capture the majority of the actual context-switch cases. The most expensive case is the one when switching between two separate applications that both make use of accelerators. In the first step we preserve (and reset) the hit/miss counters for the accelerators that were enabled in the old process. In the second step, we scan all slots and make sure that only the accelerators that should be enabled in the new process are so enabled.

The third and last call is made by *proc_free()* at process termination time, to decrement the reference count of the process' accelerators.

accelerator_switch::

```
If NoAcceleratorsInUse OR SameProcess
    return;
If PreviousProcess.AcceleratorsInUse ≠ 0
    SaveHitsMisses(PreviousProcess)
If NewProcess.AcceleratorsInUse ≠ 0
    EnableAcceleratorsFor(NewProcess)
```

Figure 7: Additions to the regular thread context switch function to collect statistics and enable the correct accelerators. Saving the statistics requires reading the miss and hit counters and incrementing the corresponding accelerator's history. This takes linear time, proportional to the number of opcodes and slots. Enabling the process' accelerators requires a linear scan of the slots to check if the accelerator loaded in a given slot is valid for the process.

We wrote two utility programs, located in the src/sbin/ directory, to support the SE image format. The ACE2SE program creates an SE image, starting from two existing software and hardware images. The SEDUMP program shows the content of an SE binary image in human-readable form. The tools currently accept the ELF format for software images and the Xilinx ACE format for accelerator images. The SE format itself is actually agnostic of these file formats, so the tools will work with any other file format.

8 Usage

Different users will want to perform one or more of the following tasks: re-building the full system from sources, install a system on a fresh disk image, either on Giano or on a board, building and loading the scheduler, and adding an accelerator to an application program. The following sections describe these tasks in details.

8.1 Building

The system is built as a cross-compilation from a NetBSD system. When using Windows as the development system, the first step is to install the VirtualPC product and create a virtual machine (and disk) for the host NetBSD system. We recommend at least 12 GB of disk space. The installation CD (ISO) images to create the host NetBSD system can be found at ftp://iso2.us.netbsd.org/pub/NetBSD/iso/<version>/ i386cd-<version>.iso, we used version 4.0.1 (and 5.99.39) without problems, but at least one other version was unable of building the system, presumably due to problems with the ACPI BIOS of the VPC. When appropriate, in the following we will refer to version 4.0.1 and related procedures.

The sources can be found on the distribution CD, and someday on the official distribution places, such as ftp://iso2.us.netbsd.org/pub/NetBSD/NetBSD-<version>/source/sets/. The following files should be installed on the host NetBSD system:

- gnusrc.tgz
- sharesrc.tgz
- src.tgz
- syssrc.tgz
- xsrc.tgz

We created the directories /usr/src, /usr/xsrc, and /usr/obj to hold sources and binaries. We made them owned by a regular user. One way to unpack the files is "for file in *.tgz; do tar -xzf $file -C /; done".

The build procedure consists of three steps: building the system, the X system, and the distribution CD images. The following commands, executed in /usr/src, will perform these three steps:

- *./build.sh -u -U -V MKSOFTFLOAT=yes -m emips release*
- *./build.sh -x -u -U -V MKSOFTFLOAT=yes -m emips release*
- *./build.sh -u -U -V MKSOFTFLOAT=yes -m emips iso-image-source*

On a current high-end portable PC these steps take approximately 14 hours, 9 hours, and a few minutes respectively. The final result is the bootable CD image /usr/obj/releasedir/iso/emipscd.iso, approximately 330 MB large. The directory /usr/obj/releasedir can be used for installation over the network. The directory /usr/obj/destdir.emips, after some manipulation, can be used as the root to mount via NFS (use for testing only).

8.2 Installation on Giano

The first step is to install the Giano simulator itself. The installation MSI file for Giano can be found at http://research.microsoft.com/en-us/downloads, look for the current Giano release page there. The second step is to create a directory to hold all the simulated system data, for instance *c:\Giano\tests\emips*. At minimum, this directory should contain the following files:

- boot.emips
- emips3.img (*)
- emipscd.iso
- Ml40x_2ace.plx
- Ml40x_bram.bin
- putty.exe (*)

All files except the starred ones can be found in the Giano installation directory. The file *boot.emips* is the NetBSD bootloader; it is built as part of building the NetBSD system for eMIPS (Section 8.1). A copy should also be on the root of the installation CD *emipscd.iso*. The file *Ml40x_bram.bin* is the primary eMIPS bootloader; it can be found and rebuilt from the official eMIPS hardware distribution. The file *Ml40x_2ace.plx* is the Giano platform configuration file that defines the Xilinx boards. This file is valid both for the ML40x and the XUP boards. The configuration adds a second SystemACE controller, useful as CD drive. The file *putty.exe* is the free terminal simulator PuTTY; it can be downloaded from http://www.chiark.greenend.org.uk/~sgtatham/putty. We use Release 0.60. The file *emips3.img* is the primary disk image for the simulated system. It should be created as empty, with the desired size, by the user. One way is to concatenate a few large files, making sure the first one is not a valid NetBSD disk image (to avoid confusion later on). For instance, you might type the command "copy/bin boot.emips+emipscd.iso+emipscd.iso emips3.img", which will create an image of approximately 700 MB. We recommend at the very least 800 MB of disk space for the system, and preferably 2 GB or more.

Assuming you have a *cmd* window open in the directory *c:\Giano\tests\emips* (or a Visual Studio Command Prompt window), start PuTTY by typing:

- *putty.exe*

and then get it ready to connect to Giano. In the configuration panel, select a *Serial* connection and set the *Serial line* name to *\\.\pipe\usart0*. It is convenient to *Save* this configuration for quick reuse later.

Next start the simulator with the following command line:

- *giano -Platform Ml40x_2ace.plx*

The simulator will initialize and pop up a warning for "Access to a non-existent memory". This is expected, software is probing I/O space to auto-configure the system. This is a good stop point that allows you to tell PuTTY to connect to the simulator. To do so, select *Open* in the configuration pane. Next go back to the warning window and select *Retry*. You should see a message on the PuTTY window from the bootloader saying "Hit any char to boot...". Do so, and you should get the following prompt:

```
NetBSD/emips <version> ...
Default: 0/ace(0,0)/netbsd
```

boot: **0/ace(1,0)/netbsd**

Answer as indicated in bold red letters, electing to boot from the installation CD. Later on you can let the system boot from the default choice. This is also selected via a timeout if you do not type anything.

This process brings up the standard NetBSD *sysinst* application. Refer to the official documentation for the various options, and for the post-install procedures: see http://www.netbsd.org/docs/guide/en/index.html. The installation CD contains documentation on the installation process also, starting with the file emips/INSTALL.html.

8.3 Installation on Boards

The installation on a Xilinx ML40x or XUP board assumes that you first create a bootable compact flash card on a PC, then download the bootloader, and finally install the bootloader into flash for operational use.

A quick way to perform the first step on NetBSD is to use dd(1) to copy the installation CD onto the compact flash card. Under Windows you can do the same using the utility *copydd.exe* from the Giano distribution. An alternate and often useful way is to tell Giano to use the compact flash drive as the primary disk drive. Assume the compact flash drive shows up as "*Disk 1"* under Disk Management. Disable all drive letters from the drive, and run the following as administrator:

- *giano -Platform Ml40x_2ace.plx SystemAce::HostFile \\.\PhysicalDrive1*

Then follow the instructions of Section 8.2 to create your bootable compact flash card.

The first time NetBSD is installed on a board you will need to download the NetBSD bootloader into RAM using the eMIPS serial line boot option. Use the *download.exe* utility from the eMIPS distribution for this. Assume the serial line connection to the board is on *Com1*. Set the dip switches for a serial line download, program the FPGA and then type the following:

- *download com1: boot.emips*

Once the download is complete, start PuTTY and tell it to make a *Serial* connection to *Serial line* COM1:, with *Speed* 38400. You will have missed the "Hit any char to boot..." prompt during the switchover, so type one char to get to the boot prompt above. Use the default boot device.

Once the system is up and running, login as root and install the bootloader into flash:

- *dd if=boot.emips of=/dev/rflash0c bs=4k conv=sync*

Note that if you used *sysinst* directly on the board it will ask you if you want it to perform this step for you, you do not need to repeat it.

Finally, change the dip switches to boot from flash (switch number zero should be set to one) and reboot. The system is now operational and should be able to boot directly from disk, without doing the download again. Note that the bootloader is also able to boot remotely via DHCP/BOOTP. Once in flash, it can be used to prime new cards directly from the net, or to boot diskless.

The BEE3 machine does not have any permanent memories, therefore on this system the bootloader must be re-downloaded on each power-up. Since there are no disks either, the only option is to run NetBSD diskless over NFS. The corresponding procedures are well known, and beyond the scope of this document.

8.4 Scheduler LKM

The scheduler is built as part of the full system build procedure (see Section 8.1), which produces two LKMs: *syscall_accel.o* and *syscall_accel_data.o.* The build places these two files in the corresponding object directories. The first is the scheduler proper; the second is a debug/maintenance interface to the scheduler. Should you need to rebuild the scheduler, in the 4.x tree, go to the source directory */usr/src/sys/lkm/syscall/accel* and cross-recompile. In a more recent version of NetBSD the location has moved to */usr/src/sys/modules*.

During installation, these two files are placed in */usr/lkm*. The scheduler is loaded like all LKMs using *modload(1*), refer to its man page for details. The following commands will do the loading:

- *modload /usr/lkm/syscall_accel.o*
- *modload /usr/lkm/syscall_accel_data.o*

Modstat(1) will verify that the LKMs are loaded properly. Look at */etc/lkm.conf(5)* to see how to enable these modules automatically. Note that *syscall_accel.o* is normally compiled with the option LOCK_THE_ICAP enabled. This creates a potential locking conflict with *dev_mkdb(1)* that can hang the system during boot. This can be solved using the AFTERMOUNT condition in the */etc/lkm.conf* entry as follows:

syscall_accel.o - - - - AFTERMOUNT

A similar locking problem arises during shutdown, because LKMs are not unloaded by default by the system. To fix this, edit the file /etc/rc.d/lkm3 to add this line:

# KEYWORD: shutdown

Failing that… you will have to halt the system manually:

shutdown now

modunload accel

halt

Two simple test programs are also built: *test_accel* and *test_accel_data*. The first manually loads an accelerator, for testing purposes. The second displays the list of all accelerators. These two programs live with the corresponding LKMs but are not normally installed in a user system. Administrators should use these example programs to build more advanced facilities instead.

8.5 Applications

This section describes how to manually add hardware acceleration to a program, using the simple test program of Appendix B as reference. The procedure is purely illustrative; other tools are normally used to automatically generate an accelerated program from an existing optimized binary program. The Giano simulator can profile and identify the blocks to accelerate; the bbtools can patch a binary and insert the extended opcodes; the M2V compiler [27] can generate the hardware accelerator from the MIPS binary code. Here we do everything manually, but for brevity we omit the creation of the hardware accelerator itself (see the manual from the eMIPS hardware distribution).

The C test program is quite simple, it reports the elapsed time taken to invoke the external function *loop(),* passing the argument *count* to it, and repeating the invocation *nloops* times. The external function itself, written in assembler, is also quite simple. The first instruction is the extended opcode to invoke the accelerator. The basic block that follows the extended opcode loops decrementing the integer argument in register *a0*, until this become zero or negative.

The idea behind this example is to create the smallest possible program that demonstrates a measurable difference between use and no-use of the accelerator. One simple implementation of the hardware accelerator, shown in Appendix B, can simply transfer control to register *ra*. The accelerator will take just one cycle to execute, the software version will execute *2+(nloops*3)* instructions instead, which will take considerable longer since eMIPS does not currently have an instruction cache.

The following command creates the optimized software binary, assuming that the C code is in the file *tloop.c* and the assembler file is in *_tloop.S*:

- *cc –O2 –o tloop tloop.c _tloop.S*

The program should be run first in this un-accelerated form, to verify that it works as expected. Since the *tloop* program image does not include or reference any accelerator, it will always run in software only.

The accelerator code is shown in Appendix B. This code is added to the standard extension boilerplate code to create a PR project, using the Xilinx ISE tools. Let us assume now that the hardware accelerator file was generated and the Xilinx tools produced the corresponding FPGA partial configuration file *tloop.bit*. In our setup the file is 109,604 bytes long. To add the accelerator to our program image we use the *ace2se* utility:

- *ace2se –ph1 tloop_a tloop tloop.bit 0 2000*

The argument *-ph1* indicates that the hardware properties should be set to 0x1. Setting bit zero of this field indicates that the accelerator plans to use opcode 24 for acceleration, which is the first of the available extended opcodes. The flags argument is 0; the accelerator does not require any special treatment. The savings argument of 2000 cycles per invocation is arbitrary since the accelerator actually provides a variable amount of speedup.

To verify that the new file *tloop_a* is indeed an accelerated application in the SE file format we can invoke the *sedump* utility:

- *sedump tloop_a*

An interesting line in the output is:

Hardware Image Properties: x1 op24

This verifies that our accelerator, if loaded, will be enabled for extended opcode 24.

Running the accelerated image will demonstrate an appreciable speedup over the un-accelerated version.

9 Conclusions

We have presented an online scheduling algorithm for hardware accelerators, its implementation on the NetBSD operating system, and an initial simple evaluation. The scheduler uses the current performance characteristics of the accelerators to select which accelerators to load and unload. The scheduler is typically within 20% of the optimal schedule computed offline. Even a much simpler greedy scheduling algorithm performed acceptably and within 30% of optimal. Differences are only notable in the cases of applications that exhibit different behaviors at different phases of their execution.

The measured overhead of the implementation on the NetBSD operating system is negligible. Using a loadable kernel module, even the code size overhead is negligible and confined to the machine-dependent potion of the kernel code. The scheduler itself compiles to 7,832 bytes of MIPS binary code.

More work is needed to improve usability and to assess the average speedups obtainable on typical user programs. The system usability is acceptable, but the tools for creating accelerators and applications, and for tuning them still need work. In our evaluations we have assumed an average application speedup of 2-4x, but we can achieve 155x with a simple TLOOP test program. This large range begs for closer investigations.

References

[1] Xilinx, Inc. Virtex 4 Family Overview. Xilinx Inc. June 2005. http://direct.xilinx.com/bvdocs/publications/ds112.pdf

[2] Forin, A., Neekzad, B., Lynch, N. L. *Giano: The Two-Headed System Simulator*. MSR-TR-2006-130, Microsoft Research, WA, September 2006.

[3] Davis, J. D., Thacker, C. P., Chang, C. *BEE3: Revitalizing Computer Architecture Research*. MSR-TR-2009-45, Microsoft Research, WA, April 2009.

[4] Sirowy, S., Forin, A. *Where's the Beef? Why FPGAs Are So Fast*. International Conference on Engineering of Reconfigurable Systems and Applications (ERSA), Las Vegas, NV, July 2009.

[5] Beeckler, J. S., Gross, W. J. *FPGA Particle Graphics Hardware*. FCCM, 2005.

[6] Tsoi, K. H., Lee, K. H., Leong, P. H. *A Massively Parallel RC4 Key Search Engine*. FCCM, 2002.

[7] Whitton, K., Hu, X. S., Yi, C. X., Chen, D. Z. *An FPGA Solution for Radiation Dose Calculation*. FCCM, 2006.

[8] Pittman, R. N., Lynch, N., Forin, A. *eMIPS, a Dynamically Extensible Processor*. MSR-TR-2006-143, Microsoft Research, WA, October 2006.

[9] Lu, H., Forin, A. *Automatic Processor Customization for Zero-Overhead Online Software Verification*. Transactions on VLSI Systems, pg. 1346-1357, November 2008.

[10] Sukhwani, B., Forin, A., Pittman, R. N. *Extensible On-Chip Peripherals*. 6th Symposium on Application Specific Processors, Anaheim, CA, June 2008.

[11] Busonera, G., Forin, A. *Exploiting partial reconfiguration for flexible software debugging.* 8th Symposium on Sytems, Architectures, Modeling and Simulation, Samos, Greece, July 2008.

[12] Sheldon, D., Vahid, F. *Making good points: application-specific pareto-point generation for design space exploration using statistical methods*. FPGA 2009, Monterey, California, USA .

[13] Borodin, A., Linial, N., Saks, M. E. *An optimal on-line algorithm for metrical task systems*. Journal of the ACM, Volume 39.4, October 1992, pp. 745 – 763.

[14] Manasse, M. S., McGeoch, L. A., Sleator, D. D. *Competitive algorithms for server problems*. Journal of Algorithms, Volume 11, Issue 2, June 1990, pg. 208-230.

[15] Karlin, A. R., Manasse, M. S., Rudolph, L., Sleator, D. D. *Competitive snoopy caching*. Algorithmica, Volume 3, 1994, pg. 79-119.

[16] Agosta, G., Palermo, G., Silvano, C. *Multi-Objective Co-Exploration of Source Code Transformations and Design Space Architectures for Low-Power Embedded Systems*. ACM Symposium on Applied Computing (SAC). 2001.

[17] Chung, E., Benini, L., De Micheli, G. *Source code transformation based on software cost analysis*. 14th International Symposium on Systems Synthesis, pg. 153-158, Montréal, Canada, 2001.

[18] Sherman, S., Baskett, F., Browne, J. C. *Trace-driven modeling and analysis of CPU scheduling in a multiprogramming system*. Communications of the ACM, Volume 15.12, pg. 1063-1069, December 1972.

[19] Lieverse, P., Van Der Wolf, P., Vissers, K., Deprettere, E. *A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems*. The Journal of VLSI Signal Processing, Volume 29.3, 2001, pg. 197-207.

[20] Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., Nicolau, A. *EXPRESSION: A Language for Architecture Exploration Through Compiler/Simulator Retargetability.* Design, Automation, and Test in Europe, pg. 31-45, 2008.

[21] FCCM. Field-Programmable Custom Computing Machines Conference. http://www.fccm.org

[22] Fu, W., Compton, K. *An Execution Environment for Reconfigurable Computing*. 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp.149-158, 2005.

[23] Huang, C., Sheldon, D., and Vahid, F. *Dynamic tuning of configurable architectures: the AWW online algorithm*. 6th International Conference on Hardware/Software Codesign and System Synthesis, Atlanta, GA, October 2008.

[24] Styles, H., Luk, W. *Compilation and management of phase-optimized reconfigurable systems*. International Conference on Field Programmable Logic and Applications, pp. 311-316, August 2005.

[25] Styles, H., Luk, W. *Exploiting Program Branch Probabilities in Hardware Compilation*. IEEE Transactions on Computers, Volume 53.11, pp. 1408-1419, November 2004.

[26] Smith, M. C. *Analytical Modeling of High Performance Reconfigurable Computers: Prediction and Analysis of System Performance*. PhD Thesis, University of Tennessee, Knoxville, 2003.

[27] Gu, R., Forin, A., Pittman, R. N. *Path-Based Scheduling in a Hardware Compiler*. International Conference on Design Automation and Test in Europe, Dresden, Germany, March 2010.

[28] Johnson, T., Shasha , D. *2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm*. 20th VLDB Conference, Santiago, Chile, 1994.

[29] Young, M., Tevenian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., Baron, R. *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.* In 11th Symposium on Operating Systems Principles. ACM, November, 1987.

[30] Liu, S., Pittman, R. N., Forin, A. *Minimizing Partial Reconfiguration Overhead with Fully Streaming DMA Engines and Intelligent ICAP Controller*. MSR-TR-2009-150, Microsoft Research, WA, September 2009.

[31] Pittman, R. N., Forin, A. *A Security Model for Reconfigurable Microcomputers*. MSR-TR-2008-121, and 3rd Workshop on Embedded Systems Security. Atlanta, GA, October 2008.

[32] Xilinx Inc. *Xilinx University Program XUPV5-LX110T Development System*, at http://www.xilinx.com/univ/xupv5-lx110t.htm

[33] Endo, Y., Wang, Z., Chen, J. B., Seltzer, M. *Using latency to evaluate interactive system performance*. SIGOPS Operating System Review, October 1996, pp. 185-199.

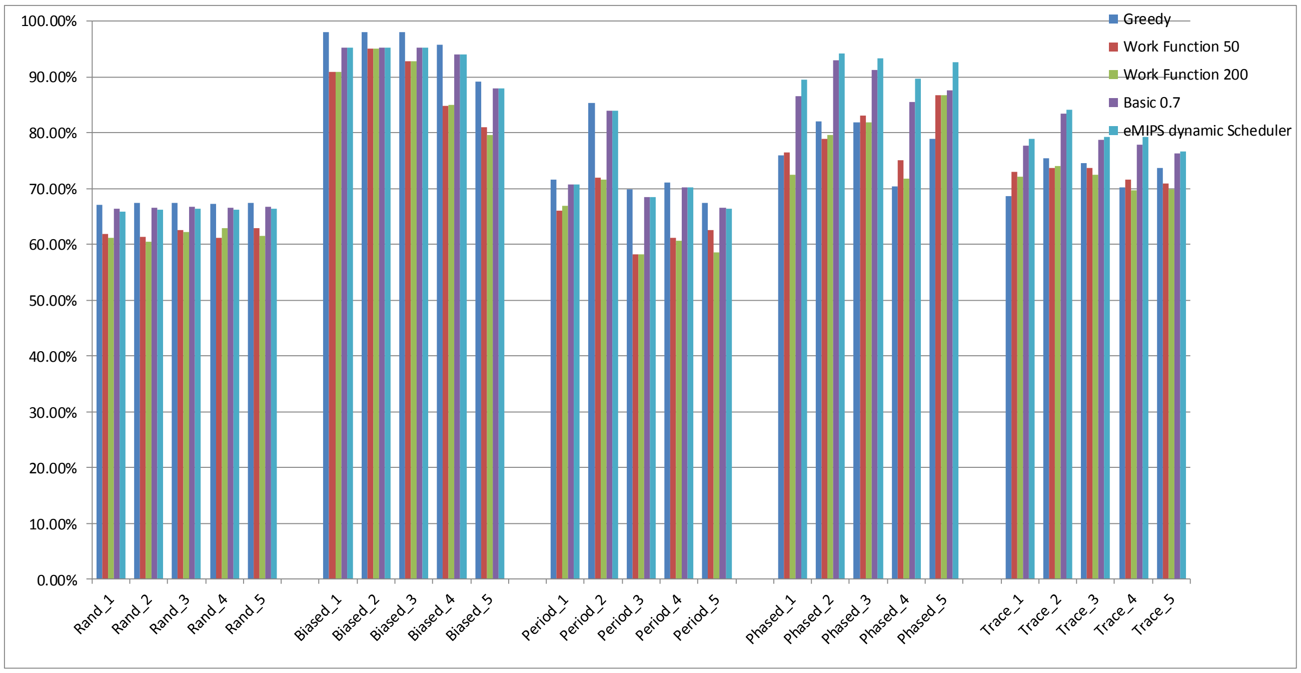

Figure 8: Performance of the five scheduling algorithms, using four different types of traces. All traces use a maximum of 7 different applications, and 4 accelerator slots. Each entry in the trace indicates that the given application was (re)activated for a (random) number of cycles. Every "random" duration is uniformly distributed between 10 and 30,000 cycles. Traces are all 100,000 events long. Loading an accelerator has a fixed cost of 112,000 cycles. The *Rand_n* traces are generated with different random seeds, with uniformly randomized application selection. In the *Biased_n* traces the application selection is biased towards a certain set. Different traces have a more remarked bias towards a smaller number of applications. The *Period_n* traces are for a set of applications with periodic activation times: the same applications repeat in the same order within each a period. The traces have periods of length 15, 15, 10, 10, and the whole trace. The *Phased_n* traces are equally subdivided in ten phases. In each phase only a given subset of applications is active, though their activation order is random. The *Trace_n* traces are a concatenation of the (corresponding) previous four traces.

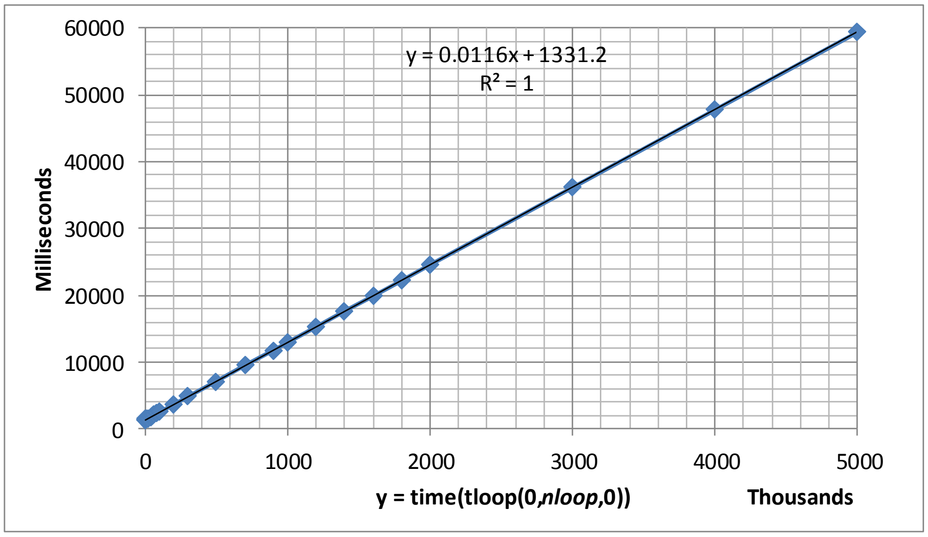

Figure 9: Behavior of TLOOP against a range of values for the *nloop* parameter, measured with the *time(1)* **facility. The other input parameters show a similar trend, as well as the measures obtained with** *gettimeofday(1)***.**

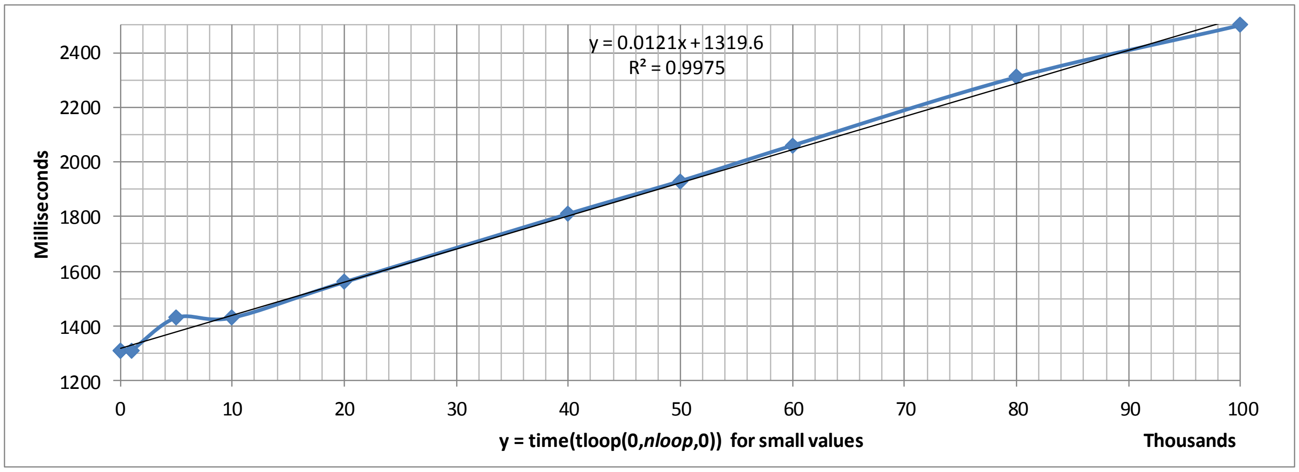

Figure 10: TLOOP is linear against *nloop* when measured by *time(1)*, except for values less than 10,000.

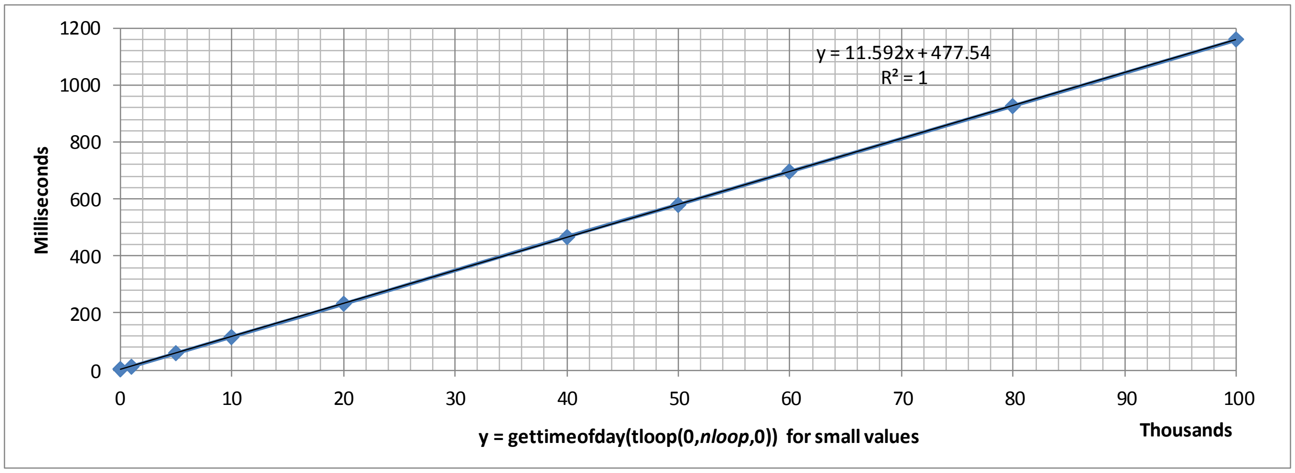

Figure 11: TLOOP is linear against all input arguments when using *gettimeofday(2)*.

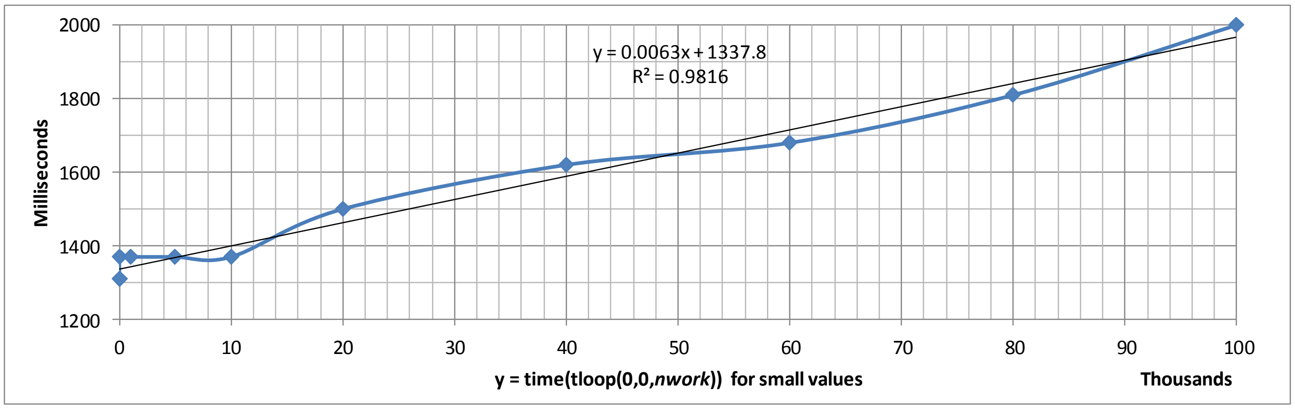

Figure 12: TLOOP is linear against *nwork* when measured by *time(1)*, except for values less than 100,000.

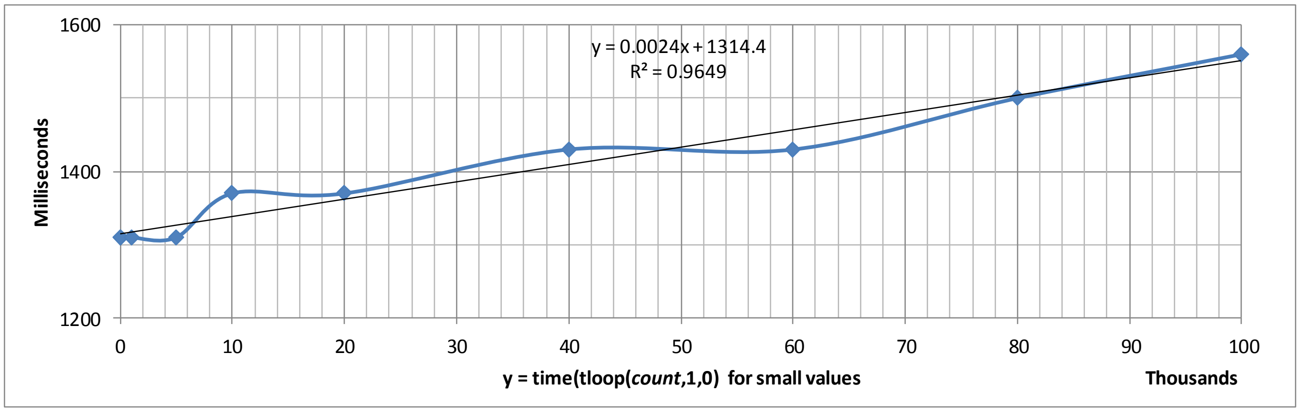

Figure 13: TLOOP is linear against *count* when measured by *time(1)*, except for values less than 80,000.

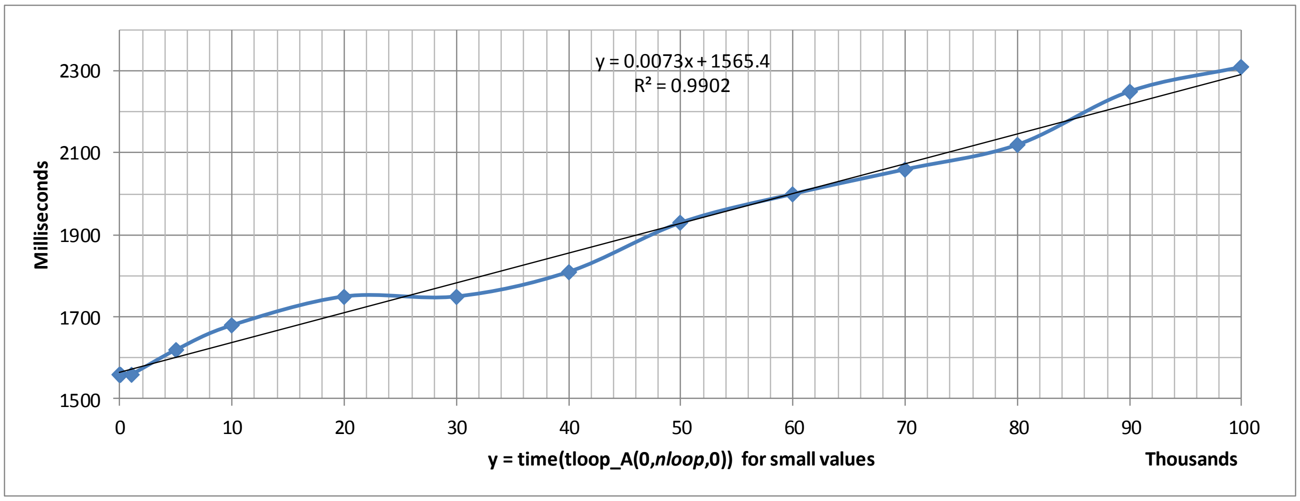

Figure 14: TLOOP_A is linear against *nloop* when measured by *time(1)*, except for values less than 100,000.

Appendix A

The following is the C code for the simulation and trace-generation program. The program can either generate one of the test traces, or execute the given scheduling algorithm on the input trace. See the function *main()* for details.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <limits.h>
#include <unistd.h>
#include <math.h>

#define TRACELEN   400000
#define NUMTASKS   7
#define NUMACC     7
#define MINRUNTIME 10
#define MAXRUNTIME 30000
#define NUMSLOTS   4
#define UPDATETIME 13000000
#define ACCLOADTIME 112000
#define INF        INT_MAX
#define BASICDECAY  .7
#define WFAWINDOW   200

int accLoadedOverride=-1;
unsigned long long int globalTime = 0;
unsigned long long int globalAccCount = 0;
unsigned long long int globalTotalTaskRuns = 0;

int readInput = 0;
int writeOutput = 0;

char inputFilename[80]  = "trace_in";
char outputFilename[80] = "trace_out";
char schedulingAlgorithm[80] = "naive";

// 0 random
// 1 biased
// 2 peridic
int traceGenerateMethod = 0;

char traceGenerateArgs[10][80];
char algArgs[5][80];


struct timestep{
   int task;
   unsigned long long int length;
};

struct wfaStruct{
   int numConfigs;
   int** combinations;
   unsigned long long int* prev;
   unsigned long long int* curr;
   int currConfig;
   int move;
   int total;
   int window;
} wfaData;

struct basic_struct{
   int accHit[NUMACC];
   int accMiss[NUMACC];
   double accSavings[NUMACC];
```

```
    int useful[NUMACC];
    int adjDecay;
    double avgDecay;
    int avgDecayCount;
} basicData;

int reconfigurations = 0;

struct timestep traceList[TRACELEN];
double taskSpeedup[NUMTASKS] = {3.96, 2.59, 3.17, 3.06, 1.12, 3.75, 2.64};
int taskSavings[NUMTASKS] = {11,21,13,11,14,14,8};
int slots[NUMSLOTS] = {-1};
unsigned long long int slotsLoadTime[NUMSLOTS] = {0};

unsigned long long int accHit[NUMACC] = {0};
unsigned long long int accMiss[NUMACC] = {0};

unsigned long long int globalAccHit[NUMACC] = {0};
unsigned long long int globalAccMiss[NUMACC] = {0};

unsigned long long int globalTaskRuns[NUMTASKS] = {0};

int fact(int n){
    if(n==1){
        return 1;
    }
    else{
        return n*fact(n-1);
    }
}

int choose(int a, int b){
    int ret = (fact(a)/( fact(a-b) * fact(b)));
    //printf("choose %d,%d = %d\n", a,b,ret);
    return ret;
}


void printc(int* comb, int k, int m){
    int i;
    printf("{");
    for(i = 0; i < k; i++){
        printf("%d, ", comb[i]);
    }
    for(i = k; i < m; i++){
        printf("%d, ", -1);
    }
    printf("\b\b}\n");
}
int next_comb(int* comb, int k, int n){
    int i = k -1;
    comb[i]++;
    while ((i >= 0) && (comb[i] >= n - k +1 + i)){
        i--;
        comb[i]++;
    }

    if (comb[0] > n - k){
        return 0;
    }

    for (i = i + 1; i < k; i++){
        comb[i] = comb[i-1]+1;
    }
    return 1;

}
```

```
void copy_array(int* dest, int* src, int size){
   int i =0;

   for (i = 0; i < size; i++){
      dest[i] = src[i];
   }
}

int** generate_combinations(int n, int k, int total){
   int** toReturn;
   int dest = 0;
   int i;
   int m = k;
   int* comb;

   //printf("generate_combinations: %d\n", total);

   comb = malloc(sizeof(int) * k);

   toReturn = malloc(sizeof(int*) * total);
   for( i = 0; i < total; i++){
      toReturn[i] = malloc(sizeof(int) * k);
   }

   for(i = 0; i < k; i++){
      comb[i] = -1;
   }
   copy_array(toReturn[dest++], comb, k);


   for(m=1; m <= k; m++){
      for(i = 0; i < k; i++){
         comb[i] = -1;
      }


      for (i = 0; i < m; i++){
         comb[i] = i;
      }

      //printc(comb,m, k);
      copy_array(toReturn[dest++], comb, k);


      while (next_comb(comb, m, n)){
         //printc(comb, m,k);
         copy_array(toReturn[dest++], comb, k);
      }
   }

   //  printf("DONE %d\n", dest);
   free(comb);
   return toReturn;
}


//task generator
void randomtask()
{
   int i;
   srand(time(NULL));
   for(i = 0; i < TRACELEN; i++)
      {
         int task;
         int length;
         task = (rand()) % NUMTASKS;
         length = rand() % MAXRUNTIME;
         if(length < MINRUNTIME){
            length = MINRUNTIME;
         }
         traceList[i].task = task;
         traceList[i].length = length;
      }
}


void biastask(){      //task generator: biased
   float a,b;
   int i, j, r;
   int k = (int)NUMTASKS * a;
   int length;

   printf("Biastask\n");

   sscanf(traceGenerateArgs[0], "%f", &a);
   sscanf(traceGenerateArgs[1], "%f", &b);


   srand(time(NULL));
   for(i = 0; i < TRACELEN; i++){

      length = rand() % MAXRUNTIME;
      if(length < MINRUNTIME){
         length = MINRUNTIME;
      }
      traceList[i].length = length;

      r = rand() % 1000;
      if(r < b * 1000){
         if(k == 0){
            j = 0;
         }
         else{
            j = rand() % k;
         }
         traceList[i].task = j;
      }
      else{
         j = rand() % NUMTASKS;
         while(j <= (NUMTASKS * a))
            j = rand() % NUMTASKS;
         traceList[i].task = j;
      }
   }
}

void periodtask(){            //task generator: period
   int a;
   int i;
   int seq[TRACELEN];
   int length;


   srand(time(NULL));

   sscanf(traceGenerateArgs[0], "%d", &a);

   for(i = 0; i < a; i++){
      seq[i] = rand() % NUMTASKS;
   }
   traceList[0].task = seq[0];
   for(i = 1; i < TRACELEN; i++){
      length = rand() % MAXRUNTIME;
      if(length < MINRUNTIME){
         length = MINRUNTIME;
      }
```

```
        traceList[i].length = length;
        traceList[i].task = seq[i % a];
    }
}

//task generator
void phasetask()
{
    int i,j;
    int numPhases;
    int numTasksPerPhase;
    int* validTasks;
    int phasedLen;
    printf("Biastask\n");

    sscanf(traceGenerateArgs[0], "%d", &numPhases);
    sscanf(traceGenerateArgs[1], "%d", &numTasksPerPhase);

    if(numTasksPerPhase > NUMTASKS){
        numTasksPerPhase = NUMTASKS;
    }
    printf("phased %d %d\n", numPhases, numTasksPerPhase);
    validTasks = malloc(sizeof(int) * numTasksPerPhase);

    srand(time(NULL));
    for(j = 0; j < numPhases; j++){
        printf("Phase %d {", j);
        for(i=0; i < numTasksPerPhase; i++){
            validTasks[i] = (rand()) % NUMTASKS;
            printf("%d ", validTasks[i]);
        }
        printf("\b}\n");
        for(i = 0; i < TRACELEN/numPhases; i++){
            int task;
            int length;
            task = validTasks[(rand()) % numTasksPerPhase];
            length = rand() % MAXRUNTIME;
            if(length < MINRUNTIME){
                length = MINRUNTIME;
            }
            traceList[((TRACELEN / numPhases) * j) + i].task = task;
            traceList[((TRACELEN / numPhases) * j) + i].length = length;
        }
    }
    /* Fill in the remainder with random */
    phasedLen = (TRACELEN / numPhases) * numPhases;
    for(i = 0; i < TRACELEN - phasedLen; i++){
        int task;
        int length;
        task = (rand()) % NUMTASKS;
        length = rand() % MAXRUNTIME;
        if(length < MINRUNTIME){
            length = MINRUNTIME;
        }
        traceList[phasedLen + i].task = task;
        traceList[phasedLen + i].length = length;
    }
}

void printTrace(){
    int i;

    for(i = 0; i < TRACELEN; i++){
        printf("Run Task %d for %lld cycles\n",
               traceList[i].task,
               traceList[i].length);
    }
}
```

```
//2 loaded but not valid
//1 true
//0 false
int isLoaded(int acc){
   if(accLoadedOverride < 0){
      int i;
      for(i = 0; i < NUMSLOTS; i++){
         if(acc == slots[i]){
            if(globalTime > slotsLoadTime[i]){
               return 1;
            }
            return 2;
         }
      }
      return 0;
   }
   else{
      return accLoadedOverride;
   }
}


void clear_counters(){
   int i;
   for(i = 0; i < NUMACC; i++){
      accHit[i] = 0;
      accMiss[i] = 0;

   }
}
void clear_global_counters(){
   int i;
   clear_counters();
   for(i = 0; i < NUMACC; i++){
      globalTaskRuns[i] = 0;
   }
   globalTime = 0;
   globalAccCount = 0;
   globalTotalTaskRuns = 0;
}

void run_scheduler_naive()
{
   int i,j;


   for(i = 0; i < NUMSLOTS; i++){
      //empty
      if(slots[i] == -1){
         int bestACC = -1;
         for(j = 0; j < NUMACC; j++){
            if(isLoaded(j) >= 1){
               continue;
            }
            if(bestACC == -1){
               if(globalAccMiss[j]*taskSavings[j] > ACCLOADTIME){
                  bestACC = j;
               }
            }
            else if(globalAccMiss[j]*taskSavings[j] >= globalAccMiss[bestACC]*taskSavings[bestACC]){
               bestACC=j;
            }
         }
         slots[i] = bestACC;
         slotsLoadTime[i] = globalTime;
      }
```

```
    }
}

//2 loaded but not valid
//1 true
//0 false
int wfa_isLoaded(int acc, int config){
    int i;
    for(i = 0; i < NUMSLOTS; i++){
        if(acc == wfaData.combinations[config][i]){
            return 1;
        }
    }
    return 0;
}

int config_diff(int old, int new){
    int i,j;
    int counter = 0;
    for(i = 0; i < NUMSLOTS; i++){
        for(j=0; j < NUMSLOTS; j++){
            if(wfaData.combinations[old][i] == wfaData.combinations[new][j]){
                if(wfaData.combinations[old][i] >= 0){
                    counter++;
                }
            }
        }
    }
    if(counter > NUMSLOTS){
        printf("CONFIG_DIFF:: MAJOR PROBLEM %d > %d\n", counter, NUMSLOTS);
        printf("{");
        for(j = 0; j < NUMSLOTS; j++){
            printf("%d ,", wfaData.combinations[old][j]);
        }
        printf("\b\b} :: ");
        printf("{");
        for(j = 0; j < NUMSLOTS; j++){
            printf("%d ,", wfaData.combinations[new][j]);
        }
        printf("\b\b}\n");
    }
    return NUMSLOTS - counter;
}

unsigned long long int wfa_runtime(int oldConfig, int newConfig, int taskno){
    unsigned long long int totalRuntime = 0;
    int configChange = 0;
    int i;


    if(oldConfig != newConfig){
        configChange = config_diff(oldConfig, newConfig);
        totalRuntime += configChange * ACCLOADTIME;
    }

    for(i = 0; i < wfaData.window; i++){
        int currTask = traceList[taskno + 1].task;
        int runTime = traceList[taskno + i].length;

        int accTime = runTime/taskSpeedup[currTask];
        //int numAccRun = (runTime-accTime)/taskSavings[currTask];


        //globalAccCount += numAccRun;
        if(wfa_isLoaded(currTask, newConfig) == 1){
            totalRuntime += accTime;
        }
        else{
            totalRuntime += runTime;
        }
    }
    return totalRuntime;
}

void init_wfa(){
    int i;
    int j;


    wfaData.numConfigs = 1;
    for(i = 1; i<=NUMSLOTS; i++){
        wfaData.numConfigs += choose(NUMACC, i);
    }
    printf("%d configs\n", wfaData.numConfigs);


    wfaData.combinations = generate_combinations(NUMACC, NUMSLOTS, wfaData.numConfigs);

    wfaData.prev = malloc(sizeof(unsigned long long int) * wfaData.numConfigs);
    wfaData.curr = malloc(sizeof(unsigned long long int) * wfaData.numConfigs);


    for(j = 0; j < wfaData.numConfigs; j++){
        wfaData.prev[j] = 0;
        wfaData.curr[j] = 0;
    }

    wfaData.currConfig = 0;
    wfaData.window = WFAWINDOW;
    /*
      for(i = 0; i < wfaData.numConfigs; i++){
      printf("{");
      for(j = 0; j < NUMSLOTS; j++){
      printf("%d ,", wfaData.combinations[i][j]);
      }
      printf("\b\b}\n");
      }
    */

}

void run_scheduler_wfa(int i){
    int j,k,s1;
    unsigned long long int min, min1;
    unsigned long long int cost;


    min1 = INF;  //min j
    for(j = 0; j < wfaData.numConfigs; j++){ // new config
        min = INF;  //min wfaData.curr[j]
        for(k = 0; k < wfaData.numConfigs; k++){ // old config k changes to j

            cost = wfaData.prev[k] + wfa_runtime(k, j, i);

            if(min > cost){
                wfaData.curr[j] = cost;
                min = cost;
            }
        }
        if(min1 > wfaData.curr[j]){
            min1 = wfaData.curr[j];
            s1 = j;
        }
    }
    for(j = 0; j < wfaData.numConfigs; j++){    //update table
```

```
        wfaData.prev[j] = wfaData.curr[j];
    }

    //above calculate loss of expert, below make online decision
    for(j = 0, min = INF; j < wfaData.numConfigs ; j++){
        if(min > wfaData.curr[j]){
            min = wfaData.curr[j];
            s1 = j;
        }
    }

    if(wfaData.currConfig != s1){
        wfaData.move++;
        wfaData.total = wfaData.total + wfa_runtime(wfaData.currConfig,s1,i);
        wfaData.currConfig = s1;
    }
    else{
        wfaData.total = wfaData.total + wfa_runtime(wfaData.currConfig,s1,i);
    }
    copy_array(slots, wfaData.combinations[wfaData.currConfig], NUMSLOTS);
}

void load_acc(int toLoad, int toUnload){
    int i;
    //printf("loading %d into %d\n", toLoad, toUnload);
    for(i = 0; i < NUMACC; i++){
        if(slots[i] == toUnload){
            slots[i] = toLoad;
            slotsLoadTime[i] = globalTime;
            return;
        }
    }
}

void run_scheduler_basic(){
    int i;
    int bestNotLoaded = -1;
    int worstLoaded = -1;
    double error[NUMACC];
    double total[2][NUMACC];
    double averageError = 0;
    double decay;
    int count = 0;
    int done = 0;

    //compute decay
    /* Determine the average % error for each accelerator in terms of
     * the past predicting this time slice
     */
    for(i=0; i<NUMACC; i++){
        total[0][i] = basicData.accHit[i] + basicData.accMiss[i];
        total[1][i] = accHit[i] + accMiss[i];
        //printf("<%d> %f = %d + %d\n", i, total[0][i], basicData.accHit[i], basicData.accMiss[i]);
        //printf("<%d> %f = %d + %d\n", i, total[1][i], accHit[i], accMiss[i]);
    }

    for(i=0; i<NUMACC; i++){
        if(total[1][i] == 0){
            error[i] = 0;
        }
        else{
            count++;
            error[i] = ((total[0][i] - total[1][i]) / total[1][i]);
            if(error[i] < 0.0){
                error[i] *= -1;
            }
        }
        //printf("error[%d] = abs((%f - %f) / %f) = %f\n", i, total[0][i], total[1][i],total[1][i], error[i]);
```

```
}
for(i=0; i<NUMACC; i++){
   averageError += error[i];
}

if(basicData.adjDecay == 1){
   decay = 1 - (averageError/NUMACC);
   if(decay < 0){
      decay = 0;
   }
   basicData.avgDecayCount++;
   basicData.avgDecay += decay;
}
else{
   //printf("using constant\n");
   decay = BASICDECAY;
}
//printf("decay = %f\n", decay);

// reduce previous value factor alpha
for(i=0; i<NUMACC; i++){
   basicData.accHit[i] = basicData.accHit[i] * decay;
   basicData.accMiss[i] = basicData.accMiss[i] * decay;
}

// add in new data
for(i=0; i<NUMACC; i++){
   basicData.accHit[i]  += accHit[i];
   basicData.accMiss[i] += accMiss[i];
}

// compute new savings
for(i=0; i<NUMACC; i++){
   basicData.accSavings[i] = basicData.accHit[i] * taskSavings[i] ;
   basicData.accSavings[i] += basicData.accMiss[i] * taskSavings[i];
}


//determine benifit of loading each acc
for(i=0; i<NUMACC; i++){
   if(isLoaded(i) == 0){
      basicData.useful[i] = basicData.accSavings[i] - ACCLOADTIME;
   }
   else{
      basicData.useful[i] = basicData.accSavings[i];
   }
}

for(i=0; i<NUMACC; i++){
   if(isLoaded(i) == 0){
      if(bestNotLoaded == -1){
         if(basicData.useful[i] > 0){
            bestNotLoaded = i;
         }
      }
      else if(basicData.useful[i] > basicData.useful[bestNotLoaded]){
         bestNotLoaded = i;
      }
   }
   else{
      if(worstLoaded == -1){
         worstLoaded = i;
      }
      else if(basicData.useful[i] < basicData.useful[worstLoaded]){
         worstLoaded = i;
      }
   }
}
```

```
    // All useable accelerators loaded
    if(bestNotLoaded == -1){
        return;
    }
    if(isLoaded(-1) > 0){
        load_acc(bestNotLoaded, -1);
    }
    else{
        if(basicData.useful[bestNotLoaded] > basicData.useful[worstLoaded]){
            load_acc(bestNotLoaded, worstLoaded);
        }
        else{
            done = 1;
        }
    }
}


void init_basic(){
    int i;
    for(i = 0; i<NUMACC; i++){
        basicData.accHit[i] = 0;
        basicData.accMiss[i] = 0;
    }
    basicData.avgDecay = 0;
    basicData.avgDecayCount = 0;
    sscanf(algArgs[0], "%d", &basicData.adjDecay);
}

int init_scheduler(){
    int i;
    for( i=0; i < NUMSLOTS; i++){
        slots[i] = -1;
    }
    if(strcmp(schedulingAlgorithm, "naive") == 0){

    }
    else if(strcmp(schedulingAlgorithm, "wfa") == 0){
        init_wfa();
    }
    else if(strcmp(schedulingAlgorithm, "basic") == 0){
        init_basic();
    }
    else{
        printf("Invalid Scheduling Algorithm: %s\n", schedulingAlgorithm);
        return 0;
    }
    return 1;
}

void wfa_cleanup(){
    /*
     int i;
     for(i = 0; i < wfaData.numConfigs; i++){
     free(wfaData.combinations[i]);
     }
     free(wfaData.combinations);
     free(wfaData.prev);
     free(wfaData.curr);
    */
}

void run_scheduler(int i){
    int j;

    // First come first serve no reconfigure
    if(strcmp(schedulingAlgorithm, "naive") == 0){
        run_scheduler_naive();
    }
    else if(strcmp(schedulingAlgorithm, "wfa") == 0){
        run_scheduler_wfa(i);
        wfa_cleanup();
    }
    else if(strcmp(schedulingAlgorithm, "basic") == 0){
        run_scheduler_basic();
    }
    else{
        printf("Invalid Scheduling Algorithm: %s\n", schedulingAlgorithm);
    }
    //printf("SLOTS: %d,%d,%d,%d\n", slots[0], slots[1], slots[2], slots[3]);
    printf("SLOTS: {");
    for(j = 0; j < NUMSLOTS; j++){
        printf("%2d,", slots[j]);
    }
    printf("\b}\n");
}

void update_counters(){
    int i;
    for (i = 0; i < NUMACC; i++){
        globalAccHit[i] += accHit[i];
        globalAccMiss[i] += accMiss[i];
    }
}

int runTrace(){

    unsigned long long int time_update = UPDATETIME;
    int i;

    clear_global_counters();

    for(i = 0; i <TRACELEN; i++){
        int currTask = traceList[i].task;
        int runTime = traceList[i].length;

        //printf("Task: %d, len: %d\n", currTask, runTime);

        globalTaskRuns[currTask]++;
        globalTotalTaskRuns++;

        int accTime = runTime/taskSpeedup[currTask];
        int numAccRun = (runTime-accTime)/taskSavings[currTask];
        //printf("numAccRun: %d\n", numAccRun);
        globalAccCount += numAccRun;
        if(isLoaded(currTask) == 1){
            //printf("Loaded\n");
            globalTime += accTime;
            accHit[currTask] += numAccRun;
        }
        else{
            //printf("Not Loaded\n");
            globalTime += runTime;
            accMiss[currTask] += numAccRun;
        }

        if(globalTime > time_update){
            time_update += UPDATETIME;

            update_counters();
            if(accLoadedOverride < 0){
                run_scheduler(i);
            }
            clear_counters();
        }
    }

    update_counters();
    printf("Total Time is: %lld :: %lld sec\n", globalTime, globalTime/7500000);
    /*
      printf("Total ACC count is: %lld\n", globalAccCount);
      printf("Hits: %lld,%lld,%lld,%lld,%lld,%lld,%lld :: %lld\n",
      globalAccHit[0],
      globalAccHit[1],
      globalAccHit[2],
      globalAccHit[3],
      globalAccHit[4],
      globalAccHit[5],
      globalAccHit[6],
      globalAccHit[0]+
      globalAccHit[1]+
      globalAccHit[2]+
      globalAccHit[3]+
      globalAccHit[4]+
      globalAccHit[5]+
      globalAccHit[6]
      );
      printf("Miss: %lld,%lld,%lld,%lld,%lld,%lld,%lld :: %lld\n",
      globalAccMiss[0],
      globalAccMiss[1],
      globalAccMiss[2],
      globalAccMiss[3],
      globalAccMiss[4],
      globalAccMiss[5],
      globalAccMiss[6],
      globalAccMiss[0]+
      globalAccMiss[1]+
      globalAccMiss[2]+
      globalAccMiss[3]+
      globalAccMiss[4]+
      globalAccMiss[5]+
      globalAccMiss[6]
      );
      printf("Task: %lld,%lld,%lld,%lld,%lld,%lld,%lld :: %lld\n",
      globalTaskRuns[0],
      globalTaskRuns[1],
      globalTaskRuns[2],
      globalTaskRuns[3],
      globalTaskRuns[4],
      globalTaskRuns[5],
      globalTaskRuns[6],
      globalTotalTaskRuns
      );
    */
    return globalTime;
}


void printTraceFile(){
    int i;
    FILE *fp;

    if(writeOutput == 0){
        return;
    }

    fp = fopen(outputFilename, "w");


    for(i = 0; i < TRACELEN; i++){
        fprintf(fp, "%d %lld\n",
                traceList[i].task,
            traceList[i].length);
    }
    fclose(fp);

}

void processCommandLine(int ARGC, char** ARGV){
    int i;

    printf("ARGC = %d\n", ARGC);

    if(ARGC == 1){
        return;
    }

    for(i = 1; i < ARGC; i++){
        // Set input file
        if(strcmp(ARGV[i], "-i") == 0){
            if(i+1>=ARGC){
                printf("ERROR: Need additional string:: %s\n", ARGV[i]);
            }
            strlcpy(inputFilename, ARGV[i+1], 80);
            i++;
        }
        // Set ouput file
        else if(strcmp(ARGV[i], "-o") == 0){
            if(i+1>=ARGC){
                printf("ERROR: Need additional string:: %s\n", ARGV[i]);
            }
            strlcpy(outputFilename, ARGV[i+1], 80);
            i++;
        }

        // Read input file
        else if(strcmp(ARGV[i], "-r") == 0){
            readInput = 1;
        }

        // Write input file
        else if(strcmp(ARGV[i], "-w") == 0){
            writeOutput = 1;
        }

        // set Algorithm
        else if(strcmp(ARGV[i], "-a") == 0){
            if(i+1>=ARGC){
                printf("ERROR: Need additional string:: %s\n", ARGV[i]);
            }
            strlcpy(schedulingAlgorithm, ARGV[i+1], 80);

            if(strcmp(schedulingAlgorithm, "basic") == 0){
                strlcpy(algArgs[0], ARGV[i+2], 80);
                i++;
            }
            i++;
        }
        // set Algorithm
        else if(strcmp(ARGV[i], "--random") == 0){
            traceGenerateMethod = 0;
        }
        // set Algorithm
        else if(strcmp(ARGV[i], "--bias") == 0){
            if(i+2>=ARGC){
                printf("ERROR: Need additional strings:: %s\n", ARGV[i]);
            }
            traceGenerateMethod = 1;
            strlcpy(traceGenerateArgs[0], ARGV[i+1], 80);
            strlcpy(traceGenerateArgs[1], ARGV[i+2], 80);
            i+=2;

        }
        // set Algorithm
        else if(strcmp(ARGV[i], "--period") == 0){
            if(i+1>=ARGC){
                printf("ERROR: Need additional strings:: %s\n", ARGV[i]);
            }
            traceGenerateMethod = 2;
            strlcpy(traceGenerateArgs[0], ARGV[i+1], 80);
            i++;
        }
        // set Algorithm
        else if(strcmp(ARGV[i], "--phase") == 0){
            if(i+2>=ARGC){
                printf("ERROR: Need additional strings:: %s\n", ARGV[i]);
            }
            traceGenerateMethod = 3;
            strlcpy(traceGenerateArgs[0], ARGV[i+1], 80);
            strlcpy(traceGenerateArgs[1], ARGV[i+2], 80);
            i+=2;
        }
        else{
            printf("ERROR: INVALID option:: %s\n", ARGV[i]);
            return;

        }

    }
}

void generateTrace(){
    printf("Generate start\n");
    if(traceGenerateMethod == 0){
        randomtask();
    }
    else if(traceGenerateMethod == 1){
        biastask();
    }
    else if(traceGenerateMethod == 2){
        periodtask();
    }
    else if(traceGenerateMethod == 3){
        phasetask();
    }
    else{
        printf("Invalid trace generation method\n");
    }
    printf("Generate end\n");
}

void read_input_file(){
    FILE *fp;
    int task, length;
    int count = 0;
    printf("Read input file\n");

    fp = fopen(inputFilename, "r");

    while(fscanf(fp, "%d %d\n", &task, &length) != EOF){
        traceList[count].task = task;
        traceList[count].length = length;
        count++;
    }
    fclose(fp);

}
```

```
void getTrace(){
   if(readInput == 1){
      read_input_file();
   }
   else{
      generateTrace();
   }
}

int main(int ARGC, char** ARGV){
   unsigned long long int timeOn, timeOff, timeAlg;//, timeOpt;
   double speedup;
   double idealSpeedup;
   double percentMax;

   printf(">>>>>>>>>>>>%d<<<<<<<<<<<<<<\n", sizeof(unsigned long long int));
   processCommandLine(ARGC, ARGV);
   init_scheduler();
   sleep(1);

   getTrace();

   //printTrace();
   printTraceFile();
   sleep(1);
   printf("ALL ON\n");
   accLoadedOverride = 1;
   timeOn = runTrace();
   timeOn = globalTime;
   sleep(1);

   printf("ALL OFF\n");
   accLoadedOverride = 0;
   timeOff = runTrace();
   timeOff = globalTime;
   sleep(1);

   printf("ALG\n");
   accLoadedOverride = -1;
   timeAlg = runTrace();
   timeAlg = globalTime;


   printf("Avg decay is %f/%d = %f\n", basicData.avgDecay, basicData.avgDecayCount, (basicData.avgDecay*1.0) / basicData.avgDecayCount);


   speedup = (timeOff*1.0)/timeAlg;
   idealSpeedup = (timeOff*1.0)/timeOn;

   percentMax = (speedup/idealSpeedup) * 100;

   printf("All On  %lld\n", timeOn);
   printf("Alg     %lld\n", timeAlg);
   printf("All Off %lld\n", timeOff);
   printf("Alg SPEEDUP of %s is %f\n", schedulingAlgorithm, speedup);
   printf("Max SPEEDUP is %f\n", idealSpeedup);
   printf("SPEEDUP of %s is %.2f%% of maximum speedup\n", schedulingAlgorithm, percentMax);

   return 0;
}
```

Appendix B

The following C program “TLOOP” invokes an external function to perform a *count* number of operations, repeating *nloops* times. A second loop, executed *nwork* times, simulates a variable amount of non-accelerated work. The elapsed time is then reported.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

extern void loop(uint32_t count);

uint64_t gettime(void)
{
   struct timeval t;
   gettimeofday(&t,NULL);
   return t.tv_usec + ((uint64_t)t.tv_sec * 1000000);
}

int main(int argc, char **argv)
{
   uint32_t count = 1, nloops = 1, nwork = 1;
   uint32_t i, w;
   uint64_t t0, t;

   if (argc > 1)
      count = atoi(argv[1]);
   if (argc > 2)
      nloops = atoi(argv[2]);
   if (argc > 3)
      nwork = atoi(argv[3]);

   t0 = gettime();
   for (i = 0; i < nloops; i++) {
      loop(count);
      for (w = 0; w < nwork; w++) {
         volatile int x = 0; x++;
      }
   }
   t = gettime() – t0;

   printf("%u loops of %u instructions took %llu usecs.\n", nloops, count*3, (uint64_t)t);
   return 1;
}
```

The following is the MIPS assembler source file for an accelerated simple loop. The extended opcode is defined by the *ExtInstruction* macro, placed right before the basic block to accelerate.

```
#include <mips/asm.h>
      .set noreorder
#define ExtInstruction(_op_,_reg1_,_reg2_,_imm_)      .int ((_op_<<26)|(_reg1_<<21)|(_reg2_<<16)|(_imm_))

LEAF(loop)
   ExtInstruction(24,4/*cosmetic*/,31/*cosmetic*/,0)
   1: bge    a0,zero,1b
   addi   a0,a0,-1
   jr     ra
   nop
END(loop)
```

The following is the Verilog source code for the TLOOP accelerator.

```
`timescale 1ns / 1ps

module tloop(
        input           CLK,            /* System Clock 50 - 100 MHZ */
        input           EN,             /* Enable */
        input [31:0]    RDREG2DATA,     /* Register Read Port 2 Register Data */
        input           RESET,          /* System Reset */
        output [31:0]   EXTADD,         /* Extension Transfer Address */
        output          RI              /* Reserved/Recognized Instruction */
        );

        reg             ri_reg;
        reg [31:0]      extadd_reg;

        initial
        begin
                ri_reg = 1'b0;
                extadd_reg = 32'b0;
        end

        assign RI       =   ri_reg;
        assign EXTADD  =    extadd_reg;


        always@(posedge CLK)
        begin
                if (RESET == 1'b0)      ri_reg <= 1'b0;
                else if (EN == 1'b0)    ri_reg <= 1'b0;
                else                    ri_reg <= 1'b1;
        end

        always@(posedge CLK)
        begin
                if (RESET == 1'b0)      extadd_reg <= 32'b0;
                else                    extadd_reg <= RDREG2DATA;
        end

endmodule
```