Dynamic Coprocessor Management for FPGA-Enhanced Compute Platforms

Chen Huang and Frank Vahid*
Department of Computer Science and Engineering
University of California, Riverside
{chuang/vahid}@cs.ucr.edu

*also with the Center for Embedded Computer Systems, Univ. of California, Irvine

ABSTRACT

Various commercial programmable compute platforms have their processor architecture enhanced with field-programmable gate arrays (FPGAs). In a common usage scenario, an application loads custom processors into the FPGA to speed up application execution compared to processor-only execution. Transient applications, changing application workloads, and limited FPGA capacity have led to a new problem of operating-system-controlled dynamic management of the loading of coprocessors into the FPGAs for best overall performance or energy. We define the Dynamic Coprocessor Management problem and provide a mapping to an online optimization problem known as Metrical Task Systems. We introduce a robust heuristic, called the adjusting cumulative benefit (ACBenefit) heuristic, that outperforms other heuristics, including a previously developed one for MTS. For two distinct application sets, we generate numerous workloads and show that the ACBenefit heuristic to provide best results across all considered workloads. In our simulations, the heuristic's results were within 9% of the offline optimal for performance, and within 3% for energy. The heuristic may be applicable to a wide variety of dynamic architecture management problems.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Adaptable architectures, heterogeneous systems.

General Terms

Algorithm, Performance, Design.

Keywords

FPGAs, dynamic optimization, runtime configuration, coprocessing, acceleration, online algorithms.

1. INTRODUCTION

Much research during the past decade has investigated the benefits of architectures supporting field-programmable gate arrays (FPGAs) as supplements to processors [1][14]. Several commercial computing platforms now supplement processors with FPGAs, at the system level [5][8], board level [19][16][17], and even integrated within a single chip [23][12]. As an example, the SGI Altix machine includes dozens of Itanium processors coupled with several Xilinx Virtex FPGAs, all having nearly equal access to the memory system. Benefits of FPGAs include speedups, due primarily to parallelism from the process level down to the bit level, of 10x-1000x for certain applications [1][21], such as for image processing, encryption, particle simulation, and other data-intensive parallelizable computations. Such applications may be accelerated using coprocessors executing on the FPGA that replace processor execution of critical code regions; the processor instead transfers control to the FPGA coprocessors and then waits for results.

Figure 1 shows a general reconfiguration architecture. Configuration data can be transferred to an FPGA by a specialized configuration controller. FPGA and CPU can communicate through the memory.

In most commercial products today containing FPGAs, such as TV set top boxes, medical devices, giant LED displays, or cellular base stations, the FPGA's coprocessing role is predetermined and thus static during the product's lifetime. Even in the case of dynamically reconfigurable systems, in which coprocessors are swapped in and out of the FPGA as an application executes to provide the illusion of a logically larger FPGA, the swapping schedule is usually statically determined. With the advent of FPGAs in general-purpose compute platforms, FPGAs become a resource that can be dynamically managed, akin to managing the contents of a cache memory. In particular, an application targeting an FPGA-enhanced compute platform may be written to optionally load and utilize a coprocessor on an FPGA. When the application attempts to run on the platform, if

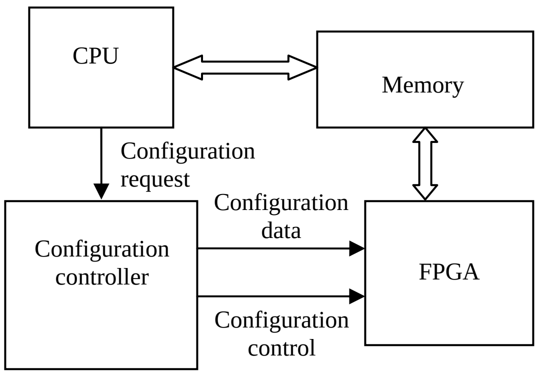


Figure 1: General reconfiguration architecture.

the application's coprocessor is not already resident in the FPGA, the platform's operating system may dynamically determine whether to run the application without the coprocessor, or to incur the overhead of loading the coprocessor into the FPGA (possibly replacing existing coprocessors to make room), a problem we refer to as *dynamic coprocessor management*.

A factor governing such management is the non-negligible time and energy cost involved in loading a coprocessor. The cost must be outweighed by the benefit for the current and/or future executions of the application. Another factor is the impact of such loading on other applications whose coprocessors are already resident in the FPGA, since removing other coprocessors to make space can have a negative impact on future executions of other applications. Without a priori knowledge of future application workloads, an OS must make decisions based on incomplete information, forming what is known as an online computing problem.

Figure 2 provides a simple example. At the current time (indicated by the arrow), the CPU has executed application *a1* twice and application *a3* once, and coprocessors *c1* and *c3* for applications *a1* and *a3* are resident in the FPGA. The queue of pending applications consists of two instances of *a2*. The problem is thus to decide whether to load *a2*'s coprocessor *c2* into the FPGA, and if so, which of *c1* or *c3* to remove to make room. If *c2* provides little speedup or its loading time is very large, it could be better to not load *c2* and instead to execute *a2* on the CPU, so that *c1* and *c3* remain resident for the future executions of *a1* and *a3*.

The main contribution of this paper is the development of an online heuristic, the *adjusting cumulative benefit heuristic*, that achieves effective dynamic coprocessor management. The heuristic outperforms other heuristics we developed, as well as a previous heuristic developed for the online optimization problem known as Metrical Task Systems. The heuristic may be applicable to a variety of dynamic architecture management problems. The paper first defines the problem, introduces various heuristics, and describes experiments. The experiments show the adjusting cumulative benefit heuristic to consistently outperform other heuristics, and to be robust in the presence of different application workload scenarios.

2. PROBLEM DEFINITION

We assume that applications written for a particular compute platform (e.g., for an SGI Altix machine) can include a custom coprocessor design for the platform's FPGA. The application may run entirely on the platform's processor, or it may run using the coprocessor on the FPGA to speed up the application's execution. Throughout most of this paper, we refer to processor, FPGA, and coprocessor in the singular. However, such reference includes situations where the platform contains multiple processors on which a single application may run, where the platform contains multiple FPGAs that are logically treated as one large FPGA, and where multiple coprocessors exist for a single application (which we collectively refer to as one coprocessor for the application). Such reference also includes the situation where a coprocessor entirely executes an application. Presently, programming systems with FPGA coprocessors uses techniques custom to each platform. Standardized techniques are an area of research [10].

We define the dynamic coprocessor management problem as follows. Given are:

- An application set *A= {a1, a2, a3, ...an}* containing the *n* application types that will run on the platform.

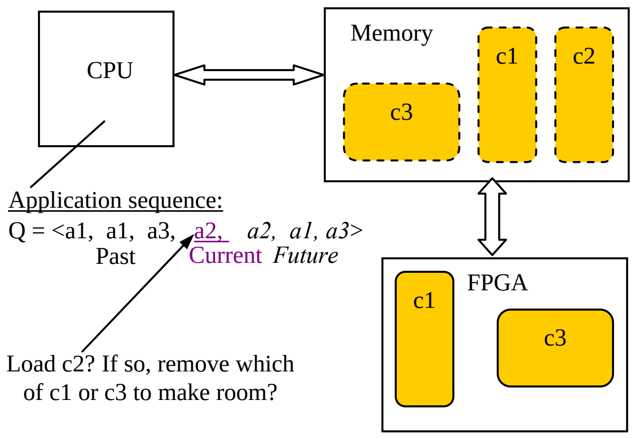

CPU
Memory
c1
c2
c3
Application sequence:
Q = <a1, a1, a3, a2, a2, a1, a3>
Past
Current
Future
FPGA
c1
c3
Load c2? If so, remove which of c1 or c3 to make room?

Figure 2: Dynamic coprocessor management: Given the sequence of past applications and current queue of pending applications, but without knowledge of future applications, determine whether to load the current application's non-resident coprocessor and which resident coprocessor(s) to remove (if necessary), such that total execution time (including loading time) of the *entire* sequence (including the future) is minimized.

- A set of execution times *Tp={tp1, tp2, tp3,…, tpn}* containing the execution time of each application type *i* on the platform's processor only.
- A set of execution times *Tc={tc1, tc2, tc3,…,tcn}* for each application type *i* when the application's coprocessor is FPGA-resident (meaning the coprocessor is in the FPGA) and utilized. These times include any additional communication times introduced by dividing an application between processor and FPGA, excluding reconfiguration time.
- A set of energies *Ep={ep1, ep2, ep3,…, epn}* giving the energy for each application type running on the platform's processor only.
- A set of energies *Ec={ec1, ec2, ec3,…, ecn}* giving the energy for each application type running when the application's coprocessor is FPGA-resident.
- A set of sizes *S={s1 ,s2, s3,…, sn}* giving the size of each application type's coprocessor in terms of equivalent gates in the FPGA.
- The total size capacity *SF* of the FPGA, in equivalent gates.
- The time for reconfiguration *TR* per gate of the FPGA, from which the total coprocessor loading time, written as *loading time(i)=TR*si*, can be computed for a coprocessor of a given size. Unloading a coprocessor takes negligible time, as it consists merely of invalidating an FPGA region.
- The energy for reconfiguration *ER* per gate of the FPGA, from which *loading energy(i)=ER*si* can be computed for a coprocessor of a given size.

The dynamic input to the problem is an application queue *Q*, such as *<a2, a1, a4, a2, a1, a1….>* that lists and orders the application instances that run on the platform.

The dynamic coprocessor management (DCM) problem for time is defined as an online problem: For each application in the application queue, using only knowledge of *prior* and *current* applications in the queue, determine whether to load that application's coprocessor, such that time for the *entire* queue (including future applications) is minimized. When a coprocessor is in the FPGA, we refer to the coprocessor as being *FPGA resident*. The *current application* is the application that at a given time is to be executed next and for which the coprocessor load determination must be made. Thus, the solution to the DCM problem consists of a coprocessor management decision for each application instance in the queue. Each decision is either: load, don't load, or already loaded. For a decision to load, the decision also lists any coprocessors that are to be unloaded to make room for the new coprocessor being loaded.

An analogous DCM problem can be defined for energy minimization.

We assume that $tci < tpi$; if not, then application i is removed from consideration by the DCM problem for time minimization (likewise for *eci* and *epi* for energy minimization).

Limitations: The above problem definition has some limitations. Application execution time on a platform's processor or processors may vary depending on what other applications are simultaneously executing. An application may have several possible coprocessor configurations that tradeoff size and performance. Future work may consider multiple simultaneously executing applications and multiple coprocessor options per application. Today's FPGAs are not reconfigurable at the granularity of gates, but rather have coarser regions (e.g., stripes) that can be independently reconfigured. FPGA capacity is not solely characterized by (equivalent) gates, but also involves hard-core resources like multipliers, block RAM, and input/output pins. Future work may deal with more device-specific reconfiguration and capacity details. The problem formulation requires that an application be pre-characterized on the platform's processor and FPGA; ideally, such pre-characterization would not be required. Finally, the present formulation does not consider the possibility of running the application during reconfiguration, an improvement also left for future work.

3. HEURISTICS

3.1 Offline optimal

To determine the offline optimal solution, which will serve as the golden standard to which the online heuristics will be compared, we first map the problem to a known online problem called Metrical Task Systems (MTS). The MTS problem, defined by Borodin [7] is a well-known formulation of a class of online problems. The problem involves a task system (S,d) for processing sequences of tasks. S is a set representing states, and d is a cost matrix where $d(i, j)$ is the cost of changing from state i to state j, assumed to satisfy the triangle inequality, and assumed to have 0s on the diagonal. In a metrical system, state transition costs are symmetric, i.e., $d(i, j)$ equals $d(j, i)$. The cost of processing a task depends on the system state, and thus a task can be viewed as a vector $T=(T(1), T(2), ..., T(j))$, where $T(j)$ is the (possibly infinite) cost of processing the task while in state j. A schedule for a sequence $T1, T2,…, Tk$ of tasks is a sequence $s1, s2,…,sk$ of states where si is the state in which Ti is processed. The cost of a schedule is the sum of all task processing costs and the state transition costs incurred. An on-line scheduling algorithm is one that chooses si only knowing $T1T2…Ti$.

We can map the dynamic coprocessor management (DCM) problem to the MTS problem as follows. DCM's platform represents MTS's system. An MTS system can be configured to any state in a set of states S. Thus, in DCM, each possible subset of coprocessors in the FPGA results in a unique system configuration state (including the situation of no coprocessors in the FPGA), and hence represents MTS's set of states S. All possible subsets of the set of coprocessors could be large, but in practice is significantly reduced due to FPGA constraints (otherwise, if FPGA constraints are lax, the DCM problem is greatly simplified). Nevertheless, all possible subsets have factorial complexity, namely all possible subsets of the coprocessors, or 2^n.

For the problem mapping, each application's execution time must be specified for each system state. Because each application's execution time depends only on whether or not its coprocessor is FPGA resident, computing the time is linear with respect to the number of applications. The rest of the mapping follows straightforwardly.

An optimal solution to the MTS problem can be obtained using a dynamic programming algorithm, as described in [7]. We omit details here. The time complexity is $O(m^2)$ for each application in the application queue Q, or $O(Km^2)$ for the entire sequence, where m is the number of configurations and K is the total length of the input sequence. Note that the number of configurations m could be very large if coprocessors are small relative to FPGA capacity, approaching 2^n (all possible combinations of coprocessors in the FPGA).

3.2 A greedy heuristic

A greedy heuristic can be defined that, given a current application, always loads the application's coprocessor into the FPGA before executing the application. When the FPGA is full, the heuristic swaps out the lowest-speedup coprocessors until the FPGA capacity is sufficient for the current coprocessor. A coprocessor's speedup is defined as tpi / tci, representing the speedup obtained when implementing an application with a coprocessor versus without, ignoring the coprocessor's loading time. The time complexity for the greedy heuristic for a current application includes $O(\log n)$ to insert the new coprocessor's speedup into a sorted list of resident coprocessor speedups, plus time proportional to the number of coprocessors that must be swapped out, which is typically a small constant (but conceivably could be n).

3.3 The work function heuristic

The work function heuristic, defined in [7] for the metrical task system problem, is similar to the offline optimal dynamic programming algorithm, but only applies to an application sequence up to and including the current application in the queue only. The algorithm incrementally updates the table used in dynamic programming as each application is encountered. The time complexity of the heuristic for the current application is $O(m^2)$, where m is the number of configurations.

3.4 The cumulative benefit heuristic

We improved on the greedy heuristic as follows. The heuristic maintains a *cumulative benefit* table, containing one entry per application i. Initially, all entries are 0. When processing a current application i in the queue, the heuristic updates the cumulative benefit table for entry i using the following equation: $cbenefit(i) = cbenefit(i) + (tpi - tci)$. In other words, the $cbenefit(i)$ entry maintains the cumulative time that using a coprocessor would have saved up until this point in the application queue had that coprocessor been used for every execution of application i.

The heuristic uses the cumulative benefit to determine whether or not to load the current application's coprocessor. In particular, the heuristic chooses to load a coprocessor if the inequality *cbenefit(i) > loading_time(i)* is satisfied. This approach follows a common solution to the well-known online computing problem known as the ski-rental problem [11]. In that classic problem, a skier must decide whether to rent or purchase skis, not knowing how many times he will ski in the future. Renting is cheaper if he will ski infrequently, but purchasing is cheaper if he will ski frequently. A well-known solution with many desirable online properties is to rent until the cumulative amount spent renting equals the cost of purchasing, at which point a purchase is made. The skier is thus guaranteed to never pay more the 2x the cost of a purchase, and this approach works well for various frequency scenarios. The ski-rental and DCM problems differ, but the intuition behind the use of the above inequality satisfaction is similar.

If the FPGA lacks current capacity for coprocessor, the heuristic searches for a subset *CP* of FPGA-resident coprocessors such that removing *CP* yields sufficient FPGA capacity for the current coprocessor. The subset must satisfy the constraint that *cbenefit(i) – loading_time(i) > cbenefit(CP)*. This constraint seeks to avoid swapping out a coprocessor deemed to be of greater benefit than the current coprocessor. Finding the best subset *CP* – where best is defined as yielding the greatest difference between the left and right sides of the constraint equation above, as yielding the smallest size capable of making room for the coprocessor, or some combination thereof – is a hard problem. We currently use a greedy heuristic for finding *CP*. The heuristic adds to *CP* the FPGA-resident coprocessor having the smallest current benefit, and continues to add such coprocessors until the size of *CP*'s coprocessors equals or exceeds the size of the current coprocessor, or until *cbenefit(i) – loading_time(i) <= cbenefit(CP)*. In the former case, the current coprocessor is loaded and the coprocessors in *CP* are unloaded. In the latter case, the heuristic decides not to load the coprocessor, because doing so would require removing coprocessors deemed to be of greater benefit (and adding more coprocessors to *CP* would only further increase *CP*'s benefit). The time complexity for a current application is *O(n),* where *n* is the number of different coprocessors.

3.5 Adjusting cumulative benefit heuristic

Real application sequences tend to exhibit temporal locality – recently-executed applications are more likely to execute again in the near future than are applications that executed long ago. The cumulative benefit heuristic does not account for such locality. One way to account for temporal locality is to apply a "fading process" to the cumulative benefits table. At every step in the application queue, the process multiplies all entries in the table by a fading factor *f*, where $0 < f < 1$. Thus, if an application that executed long ago has not executed recently, its cumulative benefit value will approach 0, making it more likely to be replaced by a currently-executing application. Recently executed applications would not have been faded as much, and such executions would serve to "refresh" the cumulative benefit value too.

The choice of a good value for *f* depends in part on the overhead of reconfiguration. If reconfiguration overhead is very small, fading should be more aggressive (meaning a small *f*) because reconfigurations can be done freely without much impact, and thus current applications should be strongly preferred. On the other hand, if reconfiguration overhead is very large, fading should be more conservative (meaning a large *f*) to be sure that an earlier executed application really hasn't executed for a long time before incurring the high cost of replacing its coprocessor. We thus define *f* to be proportional to the relative overhead of reconfiguration time versus average application execution time on the processor, namely $f = min\{TR*SF/(\sum tpi/n), 1\}$.

When considering the current application, a fading process is performed for each coprocessor: *cbenefit(j) = cbenefit(j)*f* for each *j*. Then, for the current application *i*, the benefit table is updated as before, *cbenefit(i) = cbenefit(i) + (tpi – tci)*. The replacement policy is the same as the cumulative benefit heuristic.

Because this heuristic adjusts the benefit values via fading, with such fading adjusted to different reconfiguration times per gate, we refer to it as the *adjusting cumulative benefit heuristic (ACBenefit)*. The time complexity for a current application is *O(n)*. Figure 3 shows an example: There are four coprocessors (*c1*, *c2*, *c3*, *c4*). On the left is the benefit table, listing the coprocessors and their benefits; on the right is the FPGA and resident coprocessors. The shaded area is the vacant area in the FPGA.

When application *c4* arrives, there is a fading phase. *cbenefit(i)=cbenefit(i)*f* for all coprocessors. Because *tp4-tc4=100, cbenefit(c4)=cbenefit(c4)+(tp4-tc4)=140*, and *loading time(c4) = 40*, then the current *cbenefit(c4) = 140-40=100*

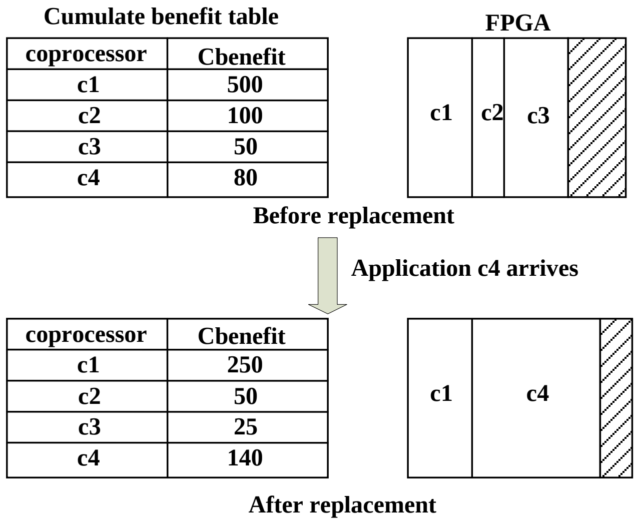
Since the coprocessor with the least benefit in the FPGA is *c3*, we swap out *c3*. Now, the current benefit is reduced to 75 by subtracting *cbenefit(c3)*. But, coprocessor *c4* still cannot be swapped in, because *c4* is still too big. Since the current benefit is larger than *cbenefit(c2),* we can further swap out coprocessor *c2,* which makes room to place coprocessor *c4*.

The adjusting cumulative benefit heuristic is shown in Figure 4.

If we want to optimize energy rather than performance, we can use the same algorithms to optimize energy instead of time.

4. EXPERIMENTS

4.1 Framework



coprocessor	Cbenefit
c1	500
c2	100
c3	50
c4	80

coprocessor	Cbenefit
c1	250
c2	50
c3	25
c4	140

Figure 3: Coprocessor replacement policy in the adjusting cumulative benefit heuristic. The fading factor f is 0.5, tp4-tc4 is 100 cycles, loading time(c4) is 40 cycles.

```
Adjusting cumulative benefit heuristic()
/*fading factor is proportional to reconfiguration time*/
$f = min\{TR*SF/(\sum tpi/n), 1\}$
For each application in Q, assuming application type is i
    For each application type j
        $cbenefit(j)=cbenefit(j)*f$   /*fading process*/
    End for
    $cbenefit(i)=cbenefit(i)+(tpi-tci)$        /*update benefit table*/
    If coprocessor i is already in FPGA
        Run the program with coprocessor
    Else  /* coprocessor i is not in the FPGA*/
        If coprocessor i can be put in FPGA
                and $cbenefit(i) > TR*si$
            Put coprocessor i in FPGA and run the program
        Else   /* no space for coprocessor i*/
            If $cbenefit(i)-TR*si > cbenefit(CP)$
                    and such coprocessors set CP exists
                Swap coprocessors CP out
                Put coprocessor i in and run the program
            Else   /*no such coprocessors CP exists */
                Run the program without coprocessor i
            End if
        End if
    End if
End for
```

Figure 5: Adjusting cumulative benefit heuristic.

We developed a simulator in C++ to test our heuristics, and applied the simulator to two benchmark sets. For each benchmark set, to evaluate the algorithms across a spectrum of application sequence scenarios, we created a generator capable of creating three categories of application sequences:

- Random: Applications are randomly inserted into the sequence.
- Biased: A small number of applications appear most of the time. We defined two percentages *A* and *B*, and then generated the sequence such that *A* percent of the applications executed *B* percent of the time. For our experiments, we used *A*=20% and *B*=80%.
- Periodic: We defined a length *T*, and generated a random subsequence of length *T* that then repeats. For our experiments, we used *T=15*.

Each sequence's length was 1,000. For all experiments, because sequences involve some random ordering, we generated 50 sequences, and report the average. For this work, execution time data does not include the time to run the heuristics themselves. For our experiments, the *ACBenefit* heuristic's runtimes were negligible relative to the benchmark and reconfiguration times. With tens of coprocessors, the heuristic required approximately 1,000 microprocessor cycles, compared to hundreds of thousands or millions of cycles of runtime for each a benchmark. The greedy and cumulative benefit heuristics' runtimes were similarly negligible. The *WorkFunction* heuristic's runtimes were also negligible due to our selected FPGA capacities only supporting a small number of coprocessors, making the number of possible configurations *m*, which determines the heuristic's runtime, small. Future work may incorporate heuristic runtime into execution time data using different benchmark runtimes and microprocessor/FPGA platforms where the heuristic runtimes may be non-negligible.

4.2 Benchmarks

Powerstone/EEMBC benchmarks: We obtained data from Stitt [22] and other sources for nine embedded system benchmarks from the Powerstone and EEMBC benchmark suites. The data is from earlier experiments seeking to show the energy advantages of partitioning applications among microprocessor and FPGA. The data included execution time, power, and size data for each benchmark, running on a microprocessor alone, or partitioned to use a coprocessor on a particular Xilinx FPGA device. Figure 5 shows the time and size data that we used. Benchmark execution time on a microprocessor (a 100 MHz MIPS processor) alone averaged 5,498 milliseconds, reduced to 2,225 milliseconds when using an FPGA, for an average speedup of 2.5x. The number of equivalent gates used on the FPGA per benchmark was 3,105, and we set the total FPGA capacity to 7,000 gates, such that coprocessor swapping would be required (a very large FPGA that

Benchmark	Orig Time (ms)	New time (ms)	Size (gates)
AIFIRF01	805	340	5770
BITMNP01	3,490	238	3393
IDCTRN01	1,500	70	2991
TTSPKR01	703	449	2759
insert	27	4	1889
binary	29	9	2232
matmul	254	26	4513
g3fax	41,974	18,836	2122
brev	701	52	2274
Average:	5,498	2,225	3,105

Figure 4: Information on EEMBC (upper case) and Powerstone (lower case) benchmarks. Original time is on a microprocessor only. New time is after partitioning frequent kernels to FPGA. Size is FPGA gates for those kernels.

can hold all or most coprocessors does not require much dynamic coprocessor management). The magnitudes of these numbers are somewhat arbitrary due to each benchmark internally being iterated a constant number of times and due to running on older microprocessor and FPGA technologies; however, the relative values of the numbers are useful for purposes of testing our heuristics.

RAW benchmarks: We obtained data from the RAW benchmark suite [2]. The suite consists of twelve programs and 37 benchmarks for comparing, validating and improving reconfigurable computing systems. We randomly chose the bheap15, bubble64, des4, fft4, Jacobi8x8, life32x6, matmult4x4, merge8, nqueens16, ssp16, and spm16 benchmarks for our experiments. Microprocessor runtimes were in the tens of milliseconds. Runtimes on FPGAs (the partitioning for these benchmarks consisted of implementing the benchmark entirely on FPGAs) averaged 10x. (Again, the execution time magnitude is somewhat arbitrary due to using older microprocessor and FPGA technologies. FPGA gate counts per benchmark averaged 48,000, and the total FPGA capacity was set to 60,000. Thus, these benchmarks required less execution time and exhibited more speedup than the Powerstone/EEMBC benchmarks, while using more gates and thus requiring more reconfiguration time.

4.3 Evaluation

Execution time: Figure 6 and Figure 7 provide results of running the various heuristics for the two benchmark suites, for the three styles of application sequences, for full-FPGA reconfiguration times ranging from 10 ms to 500 ms (thus, reconfiguration times per gate are computed by dividing the full-FPGA reconfiguration time by the total FPGA capacity). We used a range to account for a wide variety of present and future FPGA reconfiguration technologies. As can be seen, the *adjusting cumulative benefit heuristic* (*ACbenefit*) achieves results closest to optimal in nearly every scenario. In a few scenarios, the *WorkFunction* heuristic outperforms *ACbenefit*, but only very slightly; and the *WorkFunction* performs rather poorly in a couple scenarios. On average, *ACbenefit* comes within 9.2% of the offline optimal. *ACbenefit* outperforms the non-adjusting *cumulative benefit heuristic* (*Cbenefit*) when reconfiguration times are small, due to *ACbenefit* reconfiguring more frequently, because *ACbenefit* puts more weight on recent applications when reconfiguration time is small and thus when reconfigurations should be made more frequently. The *WorkFunction* heuristic performs poorly when reconfiguration times are large. When reconfiguration times are large, a heuristic should make fewer reconfigurations, but *WorkFunction* does not adjust decisions based on reconfiguration time, instead looking over the entire history.

Energy: Figure 8 shows energy results for the Powerstone benchmarks. Again, *ACbenefit* outperformed the other heuristics. *ACbenefit* was on average within 2.9% of the offline optimal. Similar results obtained for the RAW benchmarks are omitted.

Number of reconfigurations: We also recorded the number of reconfigurations incurred by each algorithm, summarized for the Powerstone benchmark suite in Figure 9. Observing the number of reconfigurations provides insight into each algorithm's behavior. The *ACbenefit* heuristic tends to match the offline optimal algorithm's number, sometimes slightly different. *WorkFunction* often performed many more reconfigurations, while still remaining competitive in total execution time. *Greedy* of course performed the most reconfigurations. *Cbenefit* heuristic usually makes much fewer reconfigurations when reconfiguration time is low, because doesn't put enough weight on the current application.

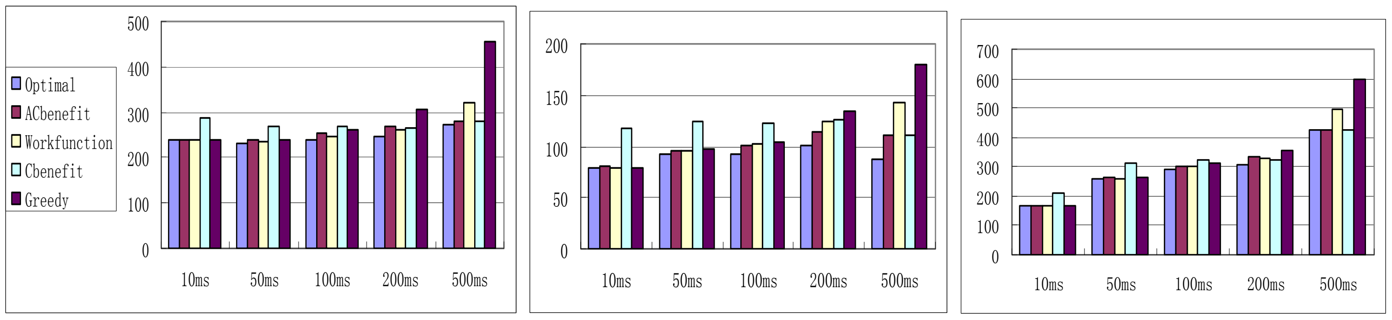


Figure 6: Powerstone/EEMBC execution times (seconds) for the various online algorithms for random (left), biased (center), and periodic (right) application sequences, for reconfiguration times for the whole FPGA ranging from 10ms to 500 ms.

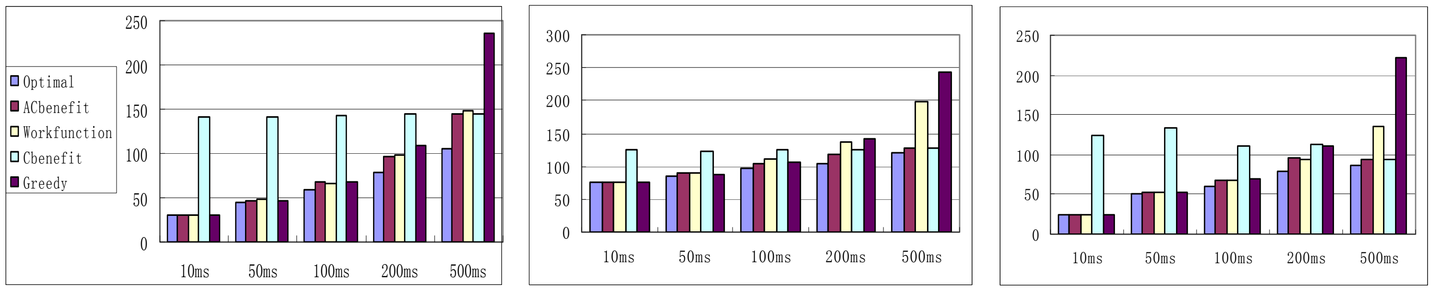


Figure 7: RAW execution times (seconds) for the various online algorithms for random (left), biased (center), and periodic (right) application sequences, for reconfiguration times for the whole FPGA ranging from 10ms to 500 ms.

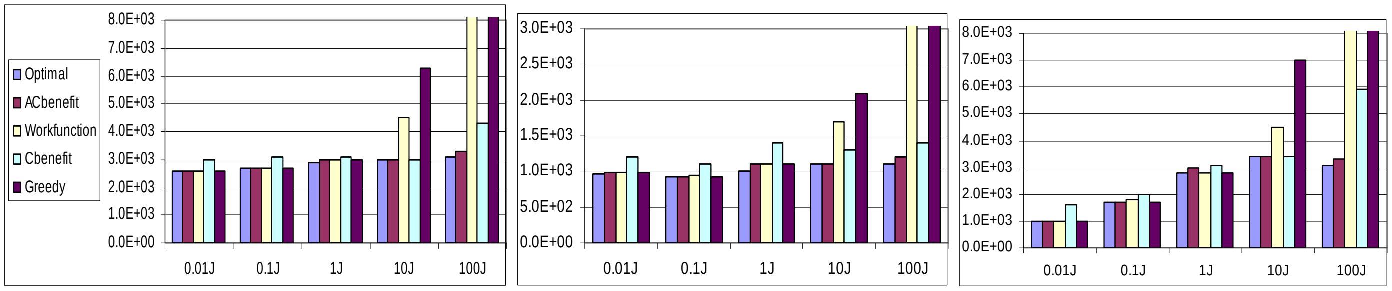


Figure 8: Powerstone/EEMBC energy consumed (J) for the various online algorithms for random (left), biased (center), and periodic (right) application sequences, for reconfiguration energy ranging from 0.01J to 100J. Certain values extend off the chart top.

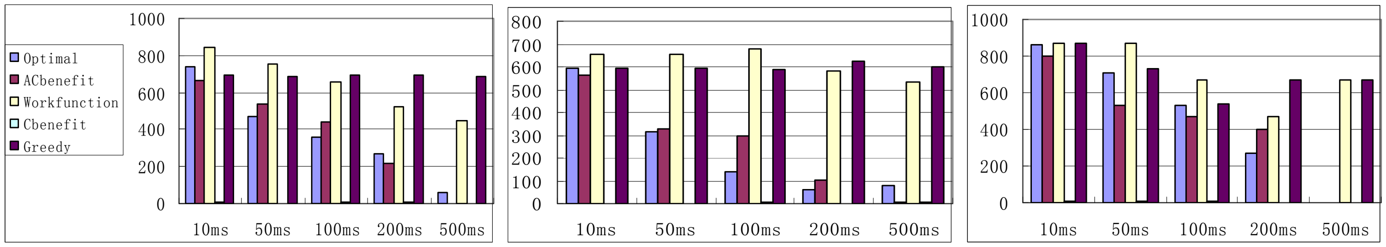


Figure 9: Powerstone/EEMBC total number of reconfigurations for random (left), biased (center), and periodic (right) application sequences, with reconfiguration times ranging from 5 ms to 200 ms.

5. RELATED WORK

Reconfiguration management for real-time embedded systems has been studied in several previous works. Balarin [3] presents a survey of real-time embedded system scheduling, which classifies the problem into static scheduling and dynamic scheduling. Lu [18] describes a static task scheduling algorithm to reorder tasks to save power in a system whose components are reconfigurable in the sense of having multiple power states. Hauck [13] proposed configuration prefetching techniques to minimize reconfiguration overhead. The idea is to load the next configuration context before it is required. Horta [15] presented a partially reconfigurable architecture in which reconfiguration is partially done within the FPGA, to reduce reconfiguration time and energy. Compton [9] proposed a relocation technique to solve the fragmentation problem of partial reconfiguration. Noguera [20] proposed dynamic run-time hardware/software scheduling techniques for FPGAs emphasizing dynamic concurrent task scheduling.

The metrical task system problem has been the focus of much online algorithm research since its definition in 1992 [7]. Many such works focus on developing K-competitive algorithms – algorithms guaranteed not to be worse than a factor of K from the offline optimal – and/or extending the problem definition (e.g. [4][6]).

6. CONCLUSIONS

We defined the dynamic coprocessor management (DCM) problem and introduced a variety of heuristics addressing the problem. We introduced a new cumulative benefit heuristic inspired by a commonly used accumulation approach in online algorithm work. We extended the heuristic to adjust to different reconfiguration times of various FPGA devices, by providing less weight to distant past items if reconfiguration time is small. The resulting adjusting cumulative benefit heuristic has linear time complexity *O(n)*, dependent only on the number of different types of applications *n*. The heuristic is more efficient than the previously-developed work function heuristic having complexity $O(m^2)$, where *m* is the number of configuration types, which is also usually much bigger than *n*. The adjusting cumulative benefit heuristic proved best in nearly all scenarios we examined, for two different benchmark sets. The heuristic's results were within 9% of the offline optimal for performance, and within 3% for energy.

7. ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation (CNS-0614957).

8. REFERENCES

[1] Altera Excalibur FPGAs, http://www.altera.com.

[2] J. Babb, M. Frank, V. Lee, E. Waingold and R. Barua. The RAW Benchmark Suite: Computation Structures for General Purpose Computing. IEEE Symposium on Field-Programmable Custom Computing 1997.

[3] F. Balarin, L. Lavagno, P. Murthy. Scheduling for Embedded Real-Time Systems. IEEE Design and Test of Computers, 1998.

[4] Y. Bartal, A. Blum, C. Burch and A. Tomkins. A polylog(n)-competitive algorithm for metrical task systems. ACM Symp. on Theory of Computing, 1997, pp. 711-719.

[5] D. Benitez. Performance of remote FPGA-based coprocessors for image-processing applications. Digital System Design, 2002.

[6] A. Blum, C. Burch. On-line Learning and the Metrical Task System Problem. Machine Learning, 2000.

[7] A. Borodin, N. Linial, and M.E. Saks. An optimal on-line algorithm for metrical task system. Journal of the ACM (JACM), Volume 39, Issue 4 (Oct. 1992), pp. 745 – 763.

[8] Celoxica, http://www.celoxica.com.

[9] K. Compton, Z. Li, J. Cooley, S. Knol and S. Hauck. Configuration relocation and defragmentation for run-time reconfigurable computing. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, 2002.

[10] J. Frigo, M. Gokhale and D. Lavenier. Evaluation of the streams-C C-to-FPGA compiler: an applications perspective. FPGA, pp. 134-140.

[11] H. Fujiwara, K. Iwama. Average-Case Competitive Analyses for Ski-Rental Problems. ISAAC 2002.

[12] S. Hauck, T.W. Fry, M.M. Hosler and J.P. Kao. The Chimaera reconfigurable functional unit. IEEE Trans. on Very Large Scale Integration (VLSI) Systems, 2004.

[13] S. Hauck. Configuration prefetch for single context reconfigurable coprocessors. Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, 1998.

[14] J.R. Hauser, J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. IEEE Symposium on FPGAs for Custom Computing Machines, 1997.

[15] E.L. Horta, J.W. Lockwood, D.E. Taylor and D. Parlour. Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration. Design Automation Conference (DAC), 2002.

[16] Intel QuickAssist Technology, http://www.intel.com/technology/magazine/45nm/quickassist-0507.htm.

[17] D. Isaacs, E. Trexel and B. Karsten. Accelerate System Performance with hybrid multiprocessing and FPGAs. Embedded Systems Design, 8/15/2007.

[18] Y. Lu, L. Benini and G. Micheli. Low Power Task Scheduling for Multiple Devices. CODES-ISSS, 2000.

[19] Mitrionics, http://www.mitrionics.com.

[20] J. Noguera, RM. Badia. Dynamic run-time HW/SW scheduling techniques for reconfigurable architectures. CODES-ISSS, 2002.

[21] SGI Altix, http://www.sgi.com/products/servers/altix/.

[22] G. Stitt, F. Vahid. Energy advantages of microprocessor platforms with on-chip configurable logic. Design & Test of Computers, IEEE, 2002.

[23] Xilinx Virtex-4 FPGAs, http://www.xilinx.com.