An Execution Environment for Reconfigurable Computing

Wenyin Fu and Katherine Compton
Department of Electrical and Computer Engineering
University of Wisconsin, Madison, WI USA
wenyinf@cae.wisc.edu, kati@engr.wisc.edu

***Abstract*— Although many studies have demonstrated the benefits of reconfigurable computing, it has not yet penetrated the mainstream. One of the biggest unsolved problems is the management of the reconfigurable hardware in a multi-threaded environment. Most research in reconfigurable computing has assumed a single-threaded model, but this is unrealistic for personal computing and many types of embedded computing. In these cases, there may be several different threads or processes running simultaneously, each wishing to use the reconfigurable hardware. The operating system must decide how to allocate the hardware at run-time based on the status of the system. The system status could also influence the choice of different implementations for each circuit based on area/speed tradeoffs. This paper examines reconfigurable computing as it could be used in mainstream systems, focusing on a proposed scheduling algorithm to allocate the reconfigurable hardware. Our initial tests indicate that reconfigurable computing with our scheduler can easily achieve at least a 20% system-level speedup.**

I. Introduction

Reconfigurable hardware is able to accelerate a variety of applications, such as DNA sequencing, MPEG, and satellite data processing [1]. Hardware is inherently more parallel than a microprocessor, and avoids the overhead of reading and decoding instructions. Reconfigurability allows the hardware to accelerate different applications and sections of a single application at different times. A variety of systems with reconfigurable hardware have been proposed and built, including Splash2 [2], PAM [3], RaPiD [4], and PipeRench [5].

Frequently, this reconfigurable hardware is used as a coprocessor to a general-purpose processor [6][7][8][9], as shown in Figure 1. The control-intensive parts of an application typically execute in software, while the compute-intensive sections are implemented in reconfigurable hardware. We refer to these latter sections as application “kernels”. Kernels are usually heavily executed loops that can be partly or completely computed in parallel. An application may have several kernels, and the hardware may be reconfigured during execution to implement each of these kernels as needed. This technique is known as reconfigurable computing (RC).

However, most work in reconfigurable computing focuses on the design of the reconfigurable logic itself or its connection to a host processor. There are a variety of problems remaining to be solved before reconfigurable hardware will be able to go mainstream as general-purpose accelerators in consumer devices. This paper presents a paradigm for the use of reconfigurable accelerators that addresses two of these issues.

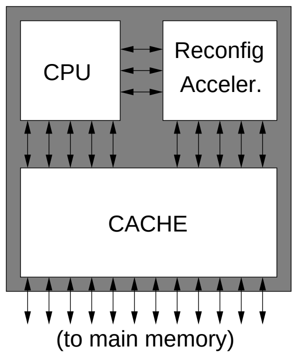

Fig. 1. Reconfigurable accelerator as a co-processor

First, application developers must be able to easily target the reconfigurable hardware. Various groups are researching compilers that can automatically detect the sections of an application that should be accelerated in hardware [12][13][14]. The goal is to create compilers that will take a program written in a high level language such as C++ or Java, and output both a binary executable and one or more configuration bitstream files for the targeted reconfigurable hardware. Currently, this depends on knowing exactly what that hardware will look like, and in many cases configuration bitstreams are not backward-compatible. This means for any upgrade of the reconfigurable logic in a system, the application must be recompiled. Support of that hardware then becomes less appealing.

Second, most previous research has assumed only a single thread of execution, where the given application has full ownership of both the host microprocessor and the reconfigurable logic. While this can be a valid assumption for application-specific and some domain-specific environments, it is not valid for more general-purpose computing and even some embedded systems, such as PDAs and cellular phones. The push towards chip multiprocessors (CMP) and simultaneous multithreaded (SMT) designs exacerbate this issue. Multiple threads from multiple applications (or even from a single application) may simultaneously demand to use the reconfigurable hardware.

To solve these two problems, we propose distributing applications both as a full software binary and with generic hardware representations of the application kernels, scheduled dynamically by the operating system. We first briefly describe the distribution method to address the first problem, then discuss the execution environment and scheduling algorithm to solve the second problem.

II. Application Distribution Model

If reconfigurable computing becomes mainstream, there will likely be multiple different hardware designs. One reason for this is varying consumer wants and needs. The graphics card industry is an example of this effect, with high-end gamers demanding significantly more powerful cards than low-end home office users. Furthermore, some consumers upgrade quickly to new designs, and others stay with what already works for them. In order for RC to be truly easy for application designers to support (critical for mainstream acceptance), they must be able to compile an application once and have it work for any of these designs.

The SCORE project [10] proposes that application design would not target a specific implementation (software, RC, ASIC). Instead, the flow of data streams is the focus. HASTE [11] proposes using a unified representation for programs which can be efficiently executed by both the CPU and the RC units. Because a special ISA must be designed to facilitate runtime hardware compilation, it is difficult to maintain binary compatibility with today's mainstream ISAs. While these are interesting directions that would solve a number of problems in RC, they would require a paradigm shift in application development that is unlikely to occur in the near- or even mid-term future. Our work therefore maintains a separation between software and hardware.

We propose that an application be distributed with both software and hardware descriptions of compute-intensive kernels. The hardware description would be in a generic low-level form such as a combination of RTL and structural Verilog, possibly encrypted to protect the vendor. Upon installation, the generic hardware description would undergo a final implementation on the target hardware. The hardware format would have to be generic enough that it could be implemented on almost any likely design, but specific enough that the final translation to actual hardware could be done quickly. The implementation phase potentially includes simple synthesis operations, mapping, placement, and routing. The driver for the reconfigurable accelerator would perform this task. The OS would call the driver as necessary in response to hardware changes (upgrades) or software installations. Applications could run "normally" immediately after install but much faster after the hardware has been fully implemented in the background, or a longer install time could provide immediate acceleration. Meanwhile, the software version could be used if reconfigurable hardware is not present, busy, or not yet configured.

Figure 2 shows an application including both a software-only binary and a set of hardware kernels. Each kernel may have one or more hardware versions at different area/speed trade-offs. In the figure, Kernel 1 has two implementations, Kernel 2 has three, and Kernel 3 has one. These hardware implementations are in addition to the software-only implementation contained in the binary executable. The multiple

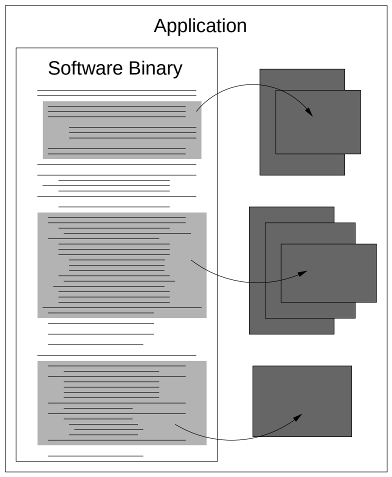

Fig. 2. Application distribution model

implementations could be created automatically during application development, or even perhaps during the application installation phase.

III. RC in a Multi-Threaded Environment

The previous section discussed application distribution and installation. We now discuss run-time issues. The key issue for RC in a multi-threaded environment is the competition between threads for the hardware. Because the applications are distributed with both software and hardware implementations of the kernels, we can decide at run-time which thread(s) should use the hardware, and which should instead run in software. Figure 3 shows an example of three different threads, T0, T1, and T2, executing in a system. The shaded areas show hardware use, whereas the light sections show execution in software. Multiple threads use the CPU and the reconfigurable hardware concurrently.

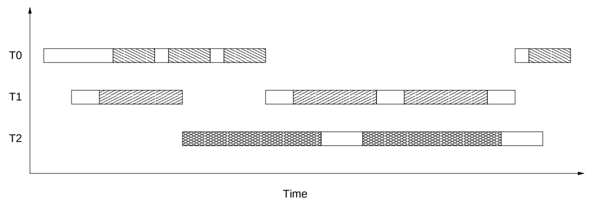

Fig. 3. Thread execution over time in SW (white) and HW (shaded)

Ideally, all threads could use the reconfigurable hardware whenever needed. However, this may not always be possible due to resource limitations. The system should not rely on the threads to schedule their own use, as invariably at least one thread will not "play fair" with device allocation. The operating system is a natural choice as independent arbiter of the reconfigurable resources [15]. We will present multiple scheduling algorithms that could be used by the OS to dynamically bind application kernels to software or hardware depending on the current system environment, set of threads, and execution phases within the threads.

The job of the scheduler is to put the reconfigurable hardware to the best use possible. This could be to accelerate the system as a whole, or to accelerate a specific compute-intense application. Note that the metric of “best use” need not be performance, and may also change over time. Perhaps while a device is powered by AC, high performance is the goal. However, when the device is powered by a battery, power consumption may be more critical. Based on the current metric, the scheduler could even choose different kernel implementations if multiple are available. We test the importance of this ability later. This choice can also be affected by the number of threads currently running in the system. If only one or a few threads are running, perhaps the fastest (but largest) implementations is appropriate. If many threads are running, smaller versions could be used to accelerate more threads, but each to a lesser degree.

As with any reconfigurable system, the configuration overhead is a significant issue, and therefore is important to the scheduling problem. The configuration overhead must not eclipse the benefit of hardware execution. While work has been done to minimize this configuration overhead both in terms of the underlying configuration hardware of the device [16] and through software techniques such as prefetching and caching of configurations [17] [18], this work has focused on a single-threaded environment. A multi-threaded environment introduces further uncertainty of the next needed configuration. The full details of how our scheduler handles configuration overhead will be presented later, but first we discuss a few assumptions about the configuration architecture and process.

To minimize configuration overhead, as well as to simplify the issues of dynamic scheduling, we have chosen a partially reconfigurable model that allows configurations to be quickly relocated to different areas of the hardware. This is admittedly a large assumption to make, as there certainly are hardware-level difficulties for this type of flexibility. However, this is not a new idea, and several techniques have been proposed.

The PRTR FPGA is a physical solution that allows for 1D relocation of both incoming configurations and those already on the hardware [16]. Alternately, a hardware abstraction solution involves dividing the hardware into tiles, where a tile represents a grouping of reconfigurable resources, and is the atomic block of reconfiguration. Configurations would be divided into a set of placed and routed tiles. When the configuration is placed onto the hardware, only the relative location of the tiles and the connections between them would need to be computed. This would allow for a fast final place-and-route pass at runtime. A physical variant of the tiling technique is to connect hardware tiles via a special bus [10]. Configuration tiles can then be placed in any physical tile without requiring very much if any run-time routing operations. We plan to further examine these ideas in future efforts.

Finally, for any reconfigurable computing system, we must examine the connection of the reconfigurable hardware to a host microprocessor. We assume the reconfigurable co-processor is attached to a host microprocessor, and is tightly coupled with the host processor’s main memory, or better, the data cache. Additionally, we must consider how the processor and reconfigurable hardware interact during scheduling and reconfiguration. For example, if we wish to avoid modifying the processor, the OS must intercept function calls to application kernels and replace them as needed with called to the reconfigurable logic. Alternately, we could extend the processor instruction set in a backward-compatible manner so the given application can also run on a system without a modified processor and reconfigurable accelerator. This type of modification would be similar to Intel’s introduction of MMX, SSE and SSE2 to its old x86 ISA. A special “branch” instruction can be added that will perform a simple low-latency table lookup to check if the kernel is presently in hardware. If so, the hardware version will be used. If not, the kernel will execute in software. Meanwhile, moving kernels into or out of hardware would remain the job of the scheduler.

IV. Scheduling and Run-Time Binding of Kernels

Scheduling of configurations onto the hardware is critical in order to maximize the hardware benefit and keep configuration overheads in check, especially as the number of hardware acceleratable kernels active in the system becomes larger. This scheduling problem has been examined by a few different research groups.

The SCORE project [10] examines dynamic scheduling, but concludes that the particular techniques used caused too much overhead. Instead, the work proposes static and quasi-static techniques. These scheduler implementations do not yet consider execution in software to be a viable alternative to hardware. Also, although the paper discusses the possibility of using a different amount of hardware resources in different execution situations, it does not appear that this situation is yet covered by the proposed scheduling techniques.

Dales described a reconfigurable resources manager, the Custom Instruction Scheduler (CIS) [19], that schedules RC hardware in a workstation environment. However, CIS (which is part of the OS), is invoked each time a kernel is called that is not present on the hardware, which could cause significant overhead. While this work does allow for both hardware and software kernel implementations, it does not consider the possibility of multiple hardware kernel implementations to allow the scheduler more flexibility in making decisions based on speed and area tradeoffs.

Before delving into our own scheduling process, we will first make an important assertion. Most programs exhibit “phases” in their execution [23]. Each phase may have its own unique program behavior, where certain kernels are invoked much more often than others. The program moves from one phase to another during execution, but inside each phase, the program’s behavior can be considered as stable. Therefore, we can consider the behavior of a program during a short time period to be approximately static. This means that we can also consider the allocation of reconfigurable hardware to be a static optimization problem.

Therefore, in all of our scheduling algorithms, we consider execution time to be divided into a sequence of time slices, known as RC scheduling intervals. At the beginning of each interval, the scheduler examines a set of candidate kernel implementations, and determines which kernels (and which implementation of each if multiple are available) should be implemented in hardware. When an application attempts to execute a kernel using the extended special branch instruction, the kernel is computed in hardware if it is already configured, or software otherwise.

The frequency of kernel use is monitored using a hardware-based scoreboard. When a kernel not yet on the scoreboard is called, it is added to the scoreboard with a score of 1. Each time the kernel is used again its scoreboard value is incremented. This scoring is performed automatically regardless of whether the kernel is used in hardware or software.

Periodically, the scoreboard is flushed to allow kernels that are no longer needed to drop off the board. If we choose to flush the scoreboard every scheduling interval, the scoreboard value for each kernel is actually its usage count starting from the last scheduling event. The scoreboard value can also be viewed as an execution frequency of the associated kernel since the last flush.

Each time RC scheduling is performed, only kernels on the scoreboard are candidates for hardware configuration. The kernel implementations determined by the scheduler to be most beneficial are loaded onto the hardware if they are not already present. The hardware maintains a table of all the kernels currently available that is updated whenever kernels are configured onto or removed from the hardware as a result of the scheduler. During the normal execution, when a thread calls a kernel, it uses the modified instruction set to conditionally execute in hardware. This instruction checks the hardware table to see if the kernel is already loaded. If so, it triggers hardware operation. If not, the kernel runs in software. Either way, the scoreboard value for that kernel is incremented as stated above.

In some cases, a kernel may be called *while* it is being configured onto the hardware. In this case, the thread will use the software version to help hide configuration latency–it does not stall waiting for configuration to complete. After the configuration completes, successive calls to the kernel will use the hardware implementation.

Next we outline several algorithms to perform kernel scheduling. These algorithms not only choose which kernels should be implemented in hardware for each scheduling interval, but also the specific hardware implementations for those kernels. Multiple hardware implementations of a kernel may be available in order to allow for a varying trade-off between speed and area. This process is used to balance the hardware resources amongst the competing threads, and choose the best combination of hardware for each scheduling interval. We first present two simple and fast-executing heuristics based on a greedy method. Next we present a scheduler that uses the Multi-Constraint Knapsack Problem (MCKP) [21] to model hardware scheduling.

A. Most Frequently Used

The benefit of the Most Frequently Used (MFU) scheduler is its simplicity and short runtime. It selects the kernels to be implemented in hardware based only on the scoreboard counter. Working from highest scoreboard value to lowest, each kernel is examined to see if its smallest implementation will fit on the available area of the reconfigurable hardware. If so, that kernel is chosen for hardware, and the area constraint is updated accordingly before considering the next kernel. This process continues until RC hardware is full or there is no valid kernel left in the candidate set. Pseudocode for the MFU algorithm is given in Figure 4.

```
let S = set of all kernels in the scoreboard;
let occupied hardware area A = 0;
while ( S is not empty && A < W_max )
  let k = kernel in S with max scoreboard count;
  let i = smallest implementation of k;
  if (A + area(i) <= W_max) then
    select i for hardware use;
    A = A + area(i);
  remove k from S;
```

Fig. 4. Pseudocode of Most Frequently Used method

B. Best Speedup

The Best Speedup algorithm uses a greedy method to choose kernels for hardware implementation based on the speedup achieved over software. In very iteration, the kernel implementation with the highest speedup gets chosen if it fits in the remaining area. This process continues until the RC hardware is full or the candidate set is empty. Pseudocode for the Best Speedup algorithm is given in Figure 5.

```
let S = set of all implementations of all
        kernels in the scoreboard;
let occupied hardware area A = 0;
while ( S is not empty && A < W_max )
  let i = implementation in S with max speedup;
  if (A + area(i) <= W_max) then
    select i for hardware use;
    A = A + area(i);
    let k = kernel implemented by i;
    remove all implementations of k from S;
  else remove i from S;
```

Fig. 5. Pseudocode of Best Speedup method

C. Multi-Constraint Knapsack

The previous two algorithms are based on simplistic greedy methods. Looking more closely at the scheduling problem, we can formulate it as follows: there are N kernels, possibly from different processes in the system, each having S_n different hardware implementations in addition to the default software implementation. The multiple hardware implementations of a kernel usually represent different trade-off points of area vs. speed or other metrics. The scheduler's problem is to allocate the limited hardware RC resources to a subset of these kernel implementations to maximize some value of the system, such as low power or high performance, while only limiting each kernel to at most one hardware implementation at a time.

Mathematically, every kernel implementation n is associated with a value $v(n,m)$ and a weight $w(n,m)$. The v represents the benefit (speedup, power reduction, etc) of using the particular hardware implementation m, while w represents the cost of using the hardware, such as the size of the configuration (area on the hardware, configuration time of the kernel). The goal is to select not only the "best" kernel(s) from the current threads but also the "best" *implementation* of those kernels during any given time slice.

Towards this end, a scheduler can attempt to maximize

$$V = \sum_{n,m} x(n,m)v(n,m), \text{ where } x(n,m) = \{0,1\}$$

while satisfying:

$$W = \sum_{n,m} x(n,m)w(n,m) \leq W_{MAX},$$

and

$$\sum_{m=1}^{S_n} x(n,m) \leq 1, n = 1, \ldots, N$$

A value 1 of $x(n,m)$ in the solution denotes that hardware version m should be executed for kernel n in this scheduling interval. The first inequality reflects the area constraint. The second one guarantees there can be at most one hardware implementation selected for each kernel. If none of the $x(n,m)$ for a particular kernel are one, it means the software implementation is selected.

This problem can be modeled by the multi-constraint knapsack problem (MCKP). Here the knapsack is the RC hardware which only has a limited area W_{MAX} (total weight the knapsack can carry), and the scheduler is trying to fill the RC resources so as to maximize V (the total value of the items). The next section discusses the MCKP and solution techniques.

1) Solving MCKP: A number of algorithms have been designed to solve the multi-constraint knapsack problem, which is known to be NP-complete. We can therefore either find an optimal solution using an NP time algorithm like dynamic programming, or use polynomial-time heuristics. However, NP computation does not necessarily translate to "prohibitively slow". The dynamic programming algorithm is in actuality a pseudo-polynomial algorithm, with a worst-case runtime of $O(S_n N W_{MAX})$. For cases where those three parameters are small, it can still run quite fast.

The scheduling algorithm runs at the beginning of every RC scheduling interval, where the interval length is T. By choosing T to be large enough, say an order of magnitude larger than the algorithm run time, we can keep the computation overhead in check. To further save the runtime, the MCKP solver will allocate the RC area in units of tiles, where a tile is some grouping of the underlying resources. In a Xilinx Virtex-series FPGA for example, a tile could consist of multiple slices so that the MCKP solver does not have to allocate area in a fine grain slice-by-slice fashion. This helps to reduce the problem size the MCKP solver must consider, again in an effort to keep the scheduler runtime reasonable. A problem with 32 kernels with 3 implementations each, and a 32-tile total hardware area has a measured scheduler runtime on a 2.0GHz Pentium 4 processor of less than 0.5ms using a dynamic programming exact solver. By setting the interval time to 0.8 seconds, the overhead can be considered negligible.

However, there are other issues to be considered when choosing an interval size T. If T is too small, both computation overhead and excessive reconfiguration can cause the system performance to be even worse than a conventional system. On the other hand, if T is made too large, optimization opportunities may be lost, and the assumption of a static problem due to program phases may become invalid. Ideally, T should match the phase duration of the programs. This means that choosing a fixed T may not be the best solution because there may be different timespans for phases both across and within processes. A dynamic detection of program phase changes and updating of T will be future work. For the work presented here, we choose a fixed T for simplicity.

2) MCKP Value Models: An intuitive way of setting v, w, and W_{MAX} in the MCKP is to let

$$v = \text{speedup} \times \text{xfreq} \tag{1}$$

where speedup is the ratio of between the hardware execution time and the default software execution time. xfreq is the number of times the kernel is called in the last RC scheduling interval which is readily available from the scoreboard. This value model focuses on fast hardware implementations for the most frequently used kernels.

However this value model does not stress the consequence of not moving a kernel to hardware, namely the software duration of the kernel. In order to see how this can affect the throughput of the system, assume we are choosing between two kernels in a given scheduling operation. The first kernel's software version has a runtime of 10K cycles, and has a hardware implementation with a 10X speedup. The second kernel's software version has a runtime of 100 cycles, and has a hardware implementation also with a 10X speedup. These two implementations will have the same value according to the above model if they are called equally frequently. However, if they are called equally frequently, implementing the first kernel in hardware will result in a greater system throughput. Our second value model attempts to capture this effect:

$$v = \text{speedup} \times \text{software xtime} \times \text{xfreq} \tag{2}$$

The last value model we present here is intended to more completely model the throughput effects of using a given implementation of a given kernel. Assume a particular kernel candidate n is currently available in implementation i, where an i value greater than 0 represents one of the hardware implementations, and i=0 represents the case where no hardware implementation is loaded and the kernel must instead execute in software. Based on this kernel's counter value in the scoreboard and the runtime of each of its implementations, we can calculate the total runtime of this kernel during the last scheduling interval to be $T_k = C \times T_i$, where C is the count and T_i is the execution time of its current implementation i. Since the OS always keeps track of the total runtime of the process, the time spent in other parts of the process (T_e) in the interval can also be obtained. Hence we now know the process spent T_k time in this kernel and T_e time outside of it.

With this breakup of the process's execution time, and assuming in the new scheduling interval the process will still get the same CPU time $T_k + T_e$, we can estimate the breakup of the time if this kernel is instead run in software in the next interval (note we already have this information if i=0) using the following two equations:

$$T'_k = (T_k + T_e) \times \frac{T_k S_i}{T_k S_i + T_e}$$

$$T'_e = (T_k + T_e) \times \frac{T_e}{T_k S_i + T_e}$$

where S_i is the speedup over software for the ith implementation. This equation states that if the kernel was previously computing in hardware, but must compute in software for the next interval, it will spend T'_k time for this kernel and T'_e time outside the kernel.

Next we consider how the different hardware implementations would affect the throughput compared to the software implementation. For implementation j of a kernel, we can estimate this process's throughput increase factor to be:

$$TPF(n,j) = \frac{T'_k + T'_e}{\frac{T'_k}{S_j} + T'_e}$$

Finally, we need to account for the fact that different threads may have been given different amounts of CPU time in the past interval by the OS scheduler (which again is different than the RC scheduler). We do this in the same way (and for the same reason) that we account for the software execution time in Eq.2. The value function for the nth kernel's jth implementation is therefore:

$$v(n,j) = TPF(n,j)(T_k + T_e) = \frac{T_k S_i + T_e}{T_k \frac{S_i}{S_j} + T_e}(T_k + T_e) \quad (3)$$

Because we are attempting to be as accurate as possible in this cost model, our implementation also accounts for the configuration time in the cost function. This makes the equation itself long and complex, but it can be easily summarized by saying that it is an adaptation of Eq.3 where if implementation j is not already on the hardware ($i \neq j$), the software version of the kernel must be used until the hardware is fully configured. We consider a possible situation where at the beginning of the interval the kernel will be called repeatedly with no intervening software computations. Worst-case, the kernel will execute in software some number of times such that the number of kernel calls times the software running time of the kernel is greater than or equal to the configuration time of the kernel. This procedure is not required if the chosen implementation is already on the hardware ($i = j$), and in other cases is only required once at the beginning of the RC scheduling interval. Note that in later sections where we indicate that our scheduler is using Eq.3, we actually mean the modified version that accounts for this configuration process.

The value functions that we have presented here target increased system throughput. However, an MCKP scheduler can be adapted to different goals by choosing different value models. Individual application performance could be emphasized, or the value model could change periodically based on the current power settings of the system using the reconfigurable hardware. For example in systems operating on battery power, the OS could optimize for power consumption by redefining the value for each kernel as:

$$v = \text{speedup} \times \text{software xtime} \times \text{xfreq}/\text{power}.$$

V. Scheduling Algorithm Simulation

To test the validity of our proposed RC paradigm, we selected three real world programs — mpeg2encode, mpeg2decode and gnupg. After profiling these applications we selected the most compute-intensive function from each (for mpeg2decode there are two equally dominant functions: idctcol and idctrow). These functions were used as our kernels. Two versions of each of these kernels were implemented in hardware on a Xilinx Virtex-II FPGA. The two different versions of the hardware provide scheduler flexibility in performing a trade-off between speed and area. In these designs, the resource requirements of the kernels are slice-bound, and so we only report the number of slices for the area.

Table I shows the details of these kernels. The software time is measured on a 2.0GHz Pentium 4 processor with 512MB RAM. The hardware time is given in terms of how many cycles the CPU goes through in the same time it takes the hardware to execute. The measure is given this way to normalize the results, and work with our value equations above. Data is given for both the "small"and "fast" versions of the kernels. For kernel "dist1", the execution time is not fixed, so the both the software and hardware execution times given in the table are the worst case values. For the other kernels, the execution time of an individual kernel is not data-dependent (though the number of times a kernel is used in an application may be). Because we are modeling a full processor load where the applications restart as soon as they complete, we not concerned at the moment with this type of data dependence.

It should be noted that these kernels were not tweaked to obtain the best performance, but were merely a quick manual conversion from C to RTL Verilog. We assume that a sophisticated compiler would be able to create configurations at least as efficient as these designs, if not more so. For very speed-critical applications, designers could almost certainly create more sophisticated implementations. The speed/area trade-off between the two different implementations of each kernel are from creating one implementation that reused hardware resources over several clock cycles, and another that pipelined the computation to achieve greater performance.

Execution traces of the kernel invocation history along with software timing information were collected for each program

Kernel	% of Program Runtime	SW time (cycles)	HW time (cycles)	Area (slices)	Kernel Speedup
idctcol	12	284	74	538	3.84
			58	811	4.90
idctrow	12	234	67	520	3.49
			51	855	4.59
dist1	42	2106	468	341	4.50
			364	653	5.79
do_encrypt	13	1243	544	162	2.28
			130	507	9.56

TABLE I

KERNEL INFORMATION: EACH KERNEL HAS A SMALL AND A FAST IMPLEMENTATION. ALL TIMES ARE GIVEN IN TERMS OF THE NUMBER OF CPU CYCLES THAT PASS DURING THE COMPUTATION TO NORMALIZE FOR DIFFERENT HARDWARE CLOCK PERIODS.

on the Pentium 4 machine mentioned above. Our custom simulator uses these traces and the area and speed information from the synthesis of the kernel implementations to evaluate the system performance using the previously-discussed scheduling techniques. The simulator uses a simple random selection method for OS scheduling decisions with a 10ms scheduling interval. The host processor modeled by the simulator supports two simultaneous running threads like the commercial Intel processors with Hyper-Threading technology[20].

Since the simulator is working with traces rather than individual instructions, it does not quite accurately model an SMT processor. Instead the model can be viewed as an idealized SMT processor where there are no conflicts between threads over all functional units and other resources. Our focus is on the RC hardware scheduling, and so we use this approximation. In future work, however, we do plan to more closely model an actual SMT processor.

Before we discuss our results, we also wish to make clear the differences between OS scheduling and RC scheduling. The former is for scheduling processes onto the threads of the processor while the latter one is responsible for allocating RC resources. In our simulator if a process is running one kernel's hardware version when an OS scheduling event happens, the OS will delay the scheduling until the current hardware kernel finishes. Since in all of our applications the hardware finishes computation very quickly (544 cycles at most), any delays in OS scheduling will be minor.

We implemented the MFU and Best Speedup greedy schedulers, as well as four different scheduling algorithms based on the multi-constraint knapsack problem. The first three MCKP implementations all use a dynamic programming method [21]. MCKP_V1 is based on the value model in Eq.1, MCKP_V2 uses the model in Eq.2, and MCKP_TP uses the value model presented in Eq.3. Finally, although we choose our RC scheduling interval to be large enough that overhead of the exact MCKP solver should not mask the benefits of using the reconfigurable hardware, we also consider a heuristic solution [21]. The MCKP_APPROX scheduler solves the MCKP by greedily choosing the kernel implementation with the best value-to-weight ratio in the candidate set while still satisfying the W_{MAX} constraint. It does this selection repeatedly until either the knapsack is full or the candidate set is empty. This MCKP heuristic uses the same value function as MCKP_TP, and therefore by comparing the two, one can examine the tradeoffs between the runtime of the exact MCKP solver and the solution quality using the heuristic.

All three applications run simultaneously on a two-way SMT simulator setting to model a heavy system load. We assume equal priority for all three processes. The RC scheduling interval T is set to 800ms. The tile size is set to be 64 slices and the configuration latency for each tile is 0.15ms which is calculated by scaling the configuration time of a typical Xilinx Virtex-II V1000 FPGA.

We run the simulator for 60 billion cycles and count the number of equivalent software cycles that each program has achieved, where each hardware kernel use contributes a number of cycles of "work" equal to the number of processor cycles the software implementation would have required. In other words, while the idctcol kernel's fast implementation may complete in hardware in 58 CPU cycles, its software execution time requires 284 cycles. Therefore, using this hardware implementation accomplishes 284 cycles of work in 58 cycles of time (remembering that hardware execution time has been recalculated in terms of CPU cycles in Table I). We sum the equivalent software cycles of work for all three programs together and divide by the number of cycles of work that would have been achieved if no reconfigurable hardware were available. This calculated value is the throughput increase factor of the system. Our calculation accounts for scheduler overhead because the scheduler execution must be performed within the examined window, and cycles spent scheduling are cycles not spent performing work, and the scheduler execution time is not considered as part of the work achieved.

Figure 6 shows the system overall throughput increase for all applications versus the available RC area for the three MCKP schedulers using the exact MCKP solver. We plot the curve for the area up to 50 tiles. At 46 tiles, all of the fastest hardware implementations (which are also the largest) of all of the kernels can fit on the hardware simultaneously. From this graph, we can see that the value model chosen for an MCKP scheduler plays an important role in the performance of the scheduler. The lack of consideration for the kernel's software duration in value model 1 severely hurts the system throughput increase. Value model 2 is actually nearly as effective as the more complex value function of Eq.3, but the more complex model does produce the best overall results.

Using the MCKP_TP scheduler (the best from Figure 6) and given a reasonable amount of RC resources, the system throughput can easily increase by more than 20%. If we reformulate this throughput increase in terms of processor speed, it is equivalent to increasing a processor from 2.0GHz to 2.4GHz. As raw clock speeds become more difficult to increase, as evidenced by Intel's recent abandonment of their latest efforts in this area [22], using reconfigurable computing becomes a more viable alternative technique to increase system performance. Also, we should emphasize that we are using very simple techniques to translate kernels to hardware, and that we are accelerating only one or two kernels from each program, as shown in TableI. With more kernels and better hardware implementations of those kernels, we would expect an even greater improvement.

In Figure 7, we contrast the performance of our best-performing MCKP scheduler, MCKP_TP, with the two simple but fast greedy methods mentioned earlier: MFU and Best Speedup. This graph also shows the result for the approximate MCKP solver (MCKP_APPROX). The scheduling overheads are accounted for in these values, so that these scheduling techniques can be more fairly compared. The simple MFU and Best Speedup heuristics do not perform nearly as well as MCKP_TP. These two heuristics only consider single metrics of frequency or speedup, and therefore have an incomplete picture of the cost/benefit of different scheduling choices. The MCKP_APPROX algorithm, also a greedy-type heuristic, performs better than MFU and Best Speedup due to the more sophisticated value function, but still performs worse than MCKP_TP. In our tests, the overhead of finding an exact solution to MCKP does not outweigh the improved solution that is found. However, as the number of kernels, number of implementations per kernel, and RC area increases, the approximate solver may become a more viable alternative, so this issue will be revisited in future work.

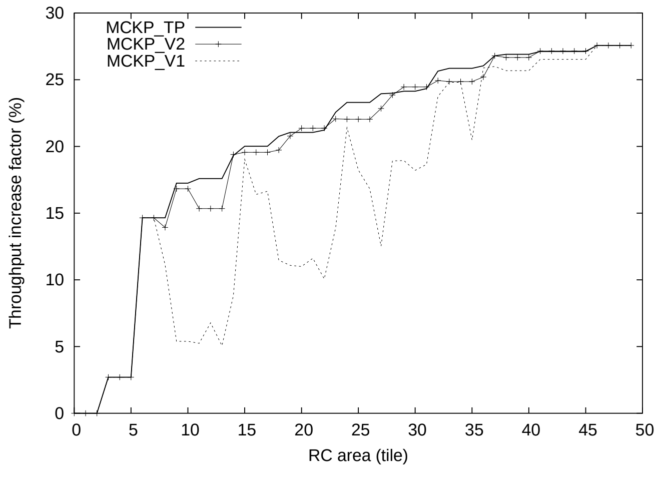

Fig. 6. Throughput increase factor vs. hardware area (I)

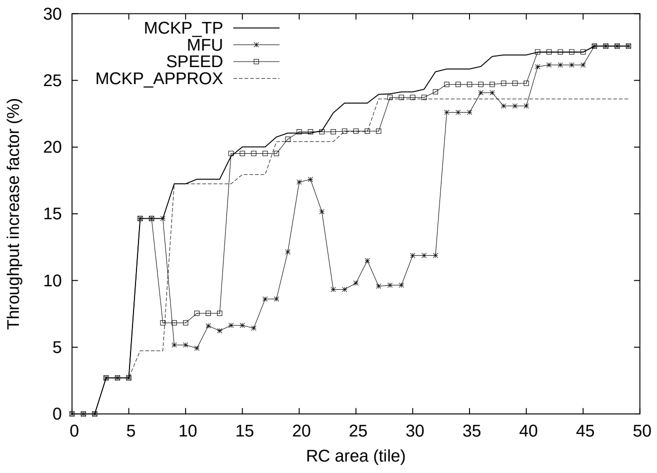

Fig. 7. Throughput increase factor vs. hardware area (II)

Thus far, we have allowed the schedulers to consider two different hardware implementations for each application kernel. Next we will test the importance of having this choice. Since the overall system throughput is strongly affected by how much of the application has been accelerated, we look only at the kernel throughput increase for this comparison in Figures 8 and 9 instead of the system throughput increase. The kernel throughput factor is defined as the ratio between the equivalent kernel cycle count and the real software-only kernel cycle count. In other words, we don't consider the non-accelerated part of the program in this graph. This is done for two reasons. First, we feel a sophisticated compiler would be able to automatically create multiple implementations the same way we did, and that these multiple implementations need not come at the cost of additional designer effort. Second, we have only accelerated a few applications, and a small portion of each, and therefore the non-accelerated code dominates in our case. For both graphs we only consider MCKP_TP scheduling algorithm. This graph shows three different curves.

In Figure 8, one curve is identical to the previous MCKP_TP results, where we allow the scheduler to choose between two implementations for each kernel. Another curve shows the case where only the fastest implementation of all kernels is available. The last curve shows the case where only the smallest implementation of all kernels is available to be scheduled. Figure 9 shows the kernel throughput with both the "Fastest only" and the "Smallest only" cases normalized against the results of MCKP_TP with both implementations available. Having a choice of implementations boosts the kernel throughput for the "middle" areas of this graph by about 30% on average.

These graphs indicate that providing multiple implementations of a single kernel greatly improves the kernel throughput factor. For the smallest hardware areas, the scheduler restricted to the smallest implementations performs better than the one restricted to the fastest (largest) implementations, and conversely, for the largest hardware areas, the "fastest only" scheduler performs better than the “smallest only” scheduler. In all cases, the scheduler with a choice of implementations provides the maximum throughput increase of the three techniques compared in the graph. With smaller areas, few if any “fastest” implementations can fit in the hardware. With larger areas, there may be room available for a faster implementation that goes unused. When the scheduler is allowed to choose the implementation, it can use different implementations based on the applications being used and the available hardware.

Having this choice is particularly important if the hardware size and complete application set are not known in advance. One such situation would be if different commercial processors were coupled with differing amounts of reconfigurable hardware. Low-end processors might include little to no reconfigurable logic, but high-end processors may have quite a bit. There would be additional hardware design issues in this situation, but for now we are only concerned with the high-level management of the hardware.

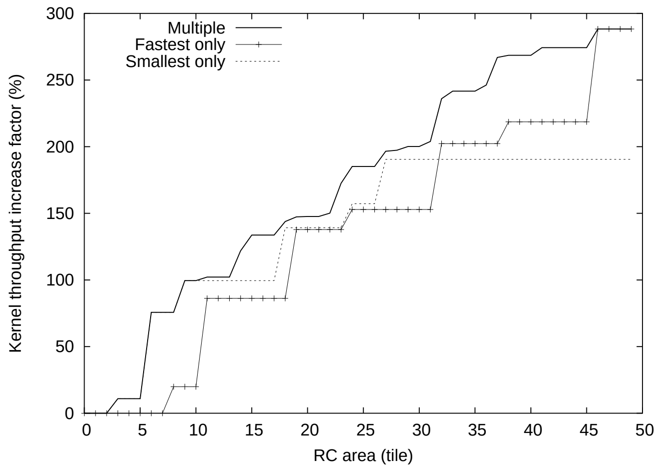

Fig. 8. Throughput increase factor when allowing multiple implementations vs. allowing only fastest/smallest ones

One hardware scheduling technique that has been suggested is to just choose one application in the system to accelerate, rather than attempt to accelerate all of them. For this next experiment, we number the applications, and each curve in Figure 10 represents the system throughput increase if only one of our three applications is accelerated. We also show the curve where, as before, we allow all applications to be accelerated. From the figure we can see that accelerating only a single application is not a good technique for increasing the overall system throughput.

We examine one of these cases, where only mpeg2decode is accelerated, in greater detail in Figure 11. This application is chosen because it has two kernels for consideration, whereas the other applications each have a single kernel. Five of our scheduling algorithms are compared in this case to test how choosing only a single application to accelerate affects the importance of the scheduling algorithm. As the graph shows, if an application has more than one kernel, the choice of scheduling algorithm can make a difference even for single-

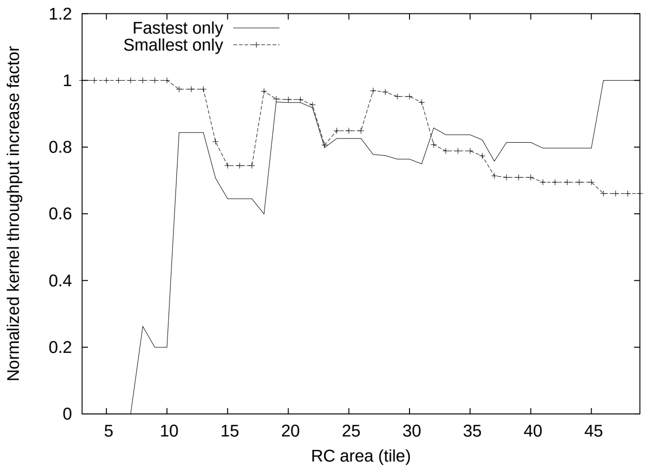

Fig. 9. Kernel throughput when only using the fastest/smallest implementations, normalized to the kernel throughput of a scheduler with a choice of multiple implementations

application acceleration.

In most previous research when accelerating a single application, the schedule of configuration loading and use was performed for each application. The application designer must (possibly with the aid of an automatic tool) perform the scheduling, and thus different applications may interact poorly when run simultaneously. Scheduling of commonly-used resources (as we hope reconfigurable hardware becomes!) belongs at a higher level than the applications. An OS-based scheduler can make scheduling decisions for the benefit of the system, and by changing the value function of the scheduler, can even emphasize the acceleration of a particular application if desired or necessary.

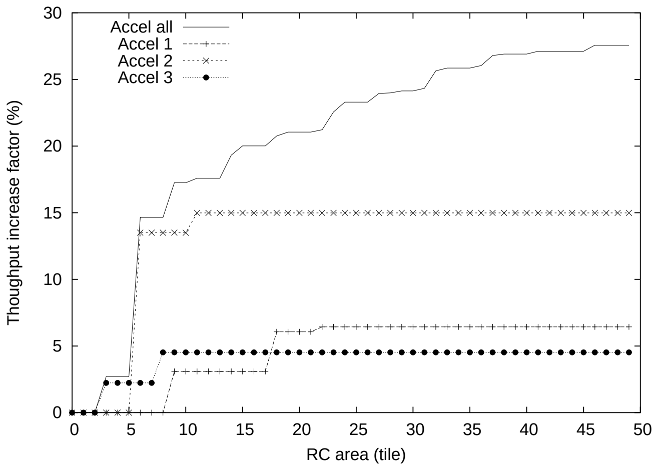

Fig. 10. Throughput increase factor when we only choose to accelerate one of the applications

VI. Future Work

There is still a wide variety of work remaining in the area of OS support for reconfigurable computing. For this particular

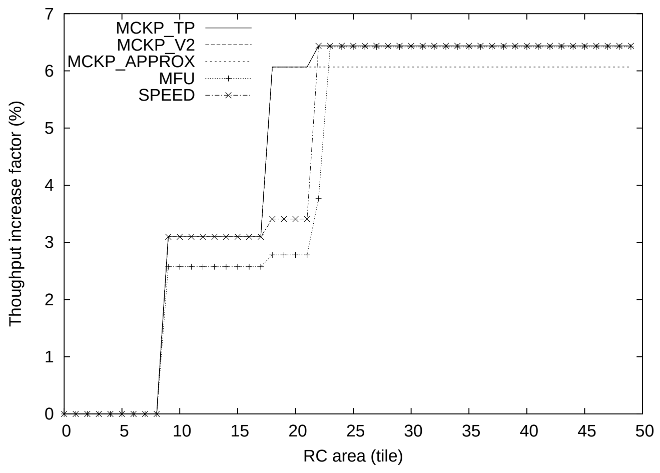

Fig. 11. Throughput increase factor when only accelerating program mpeg2decode

project, there are a number of possible future improvements. We would like to assemble a wider set of applications and hardware kernel implementations. We would also like to move towards a cycle-accurate simulation techniques that would allow us to more accurately represent an SMT processor. Furthermore, we intend to expand our testing to examine the effects of a dynamically-determined RC scheduling interval length, the effects of non-full CPU loads on the quality of our scheduling algorithms, and how our MCKP_TP value function should change if applications are allowed to have different priorities.

VII. Conclusion

This paper has focused on a proposed application distribution and execution model for reconfigurable computing. We discussed how applications supporting reconfigurable accelerators should include both the full software binary and multiple hardware implementations of each kernel. Our results show that multiple kernel implementations allow for greater overall acceleration. We also presented scheduling algorithms for allocating the reconfigurable hardware to competing threads. Threads not running in hardware are still executed in software, avoiding the traditional problem of stalling on hardware availability. Our best-performing algorithm, MCKP_TP shows an overall throughput more than 20% over software-only execution. While single-application acceleration can produce more impressive benefits of orders of magnitude, a system-level increase of 20% is significant given the increasing difficulty of achieving faster clock speeds in today's processors.

Our model for reconfigurable computing, coupled with an intelligent hardware-aware high-level-language compiler, will allow software developers to easily create hardware-accelerated applications. The distribution model and OS-level hardware scheduler will allow end-users to take advantage of hardware acceleration with no extra effort on their part. Just as importantly, our scheduler will allow for efficient use of the reconfigurable hardware in a multithreaded/multi-tasking environment. This work therefore addresses two significant problems that have kept reconfigurable computing from the mainstream.

Acknowledgments

We thank Saket Jamkar, Matthew Karlen, Arvind Prasad, and Kyle Rupnow for their help with the hardware implementation of the kernels.

References

[1] K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", *ACM Computing Surveys*, Vol. 10, No. 3, June 2002.
[2] D. Buell, J.M. Arnold, W.J. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, 1996.
[3] P. Bertin, *Memoires Actives Programmables: Conception, Realisation et Programmation*, PhD thesis, Universite Paris 7, 1993.
[4] C. Ebeling, D.C. Cronquist, P. Franklin, "RaPiD–Reconfigurable Pipelined Datapath", *Field Programmable Logic Conference*, 1996.
[5] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler", *IEEE Computer*, Vol. 33, No. 4, 2000.
[6] J.R. Hauser, J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor", *Int'l Symp. FCCM*, April 1997.
[7] T. Miyamori, K. Olukotun, "REMARC: Reconfigurable Multimedia Array Coprocessor", *Proc. Int'l Symp. FPGA*, February 1998.
[8] Z.A. Ye, A. Moshovos, S. Hauck, P. Banerjee, "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit", *Proc. Int'l Symp.Computer Architecture*, June 2000.
[9] R.D. Wittig, P. Chow, "OneChip: An FPGA Processor With Reconfigurable Logic", *Int'l Symp. FCCM*, April 1996.
[10] E. Caspi, R. Huang, Y. Markovskiy, J. Yeh, J. Wawrzynek, A. DeHon, "A Streaming Multi-Threaded Model", *Third Workshop on Media and Stream Processing (MSP-3)*, December 2001.
[11] B. Levine, H. Schmit, "Efficient application representation for HASTE: Hybrid Architectures with a Single, Transformable Executable", *Int'l Symp. FCCM*, April 2003.
[12] R. Allen, D. Gajski, "The Case for C/C++ Hardware Design", *EEDesign*, June 9, 2000.
[13] T. Callahan, J.R. Hauser, J. Wawrzynek, "The Garp Architecture and C Compiler", *IEEE Computer*, April 2000.
[14] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, J. Stockwood, "Hardware-Software Co-Design of Embedded Reconfigurable Architectures", *Design Automation Conference*, 2000.
[15] G. Brebner, "A Virtual Hardware Operating System for the Xilinx XC6200", *FPL* 1996.
[16] K. Compton, Z. Li, J. Cooley, S. Knol, S. Hauck, "Configuration Relocation and Defragmentation for Run-Time Reconfigurable Systems", *IEEE Trans. on VLSI*, Vol. 10, No. 3, June 2002.
[17] Z. Li, *Configuration Management Techniques for Reconfigurable Computing*, Ph.D. Thesis, Northwestern University.
[18] B. Xu, D.H. Albonesi, "Runtime Reconfiguration Techniques for Efficient General-Purpose Computation", *IEEE Design and Test*, Vol.17 No.1, p.42-52, January 2000.
[19] M. Dales, *Managing a Reconfigurable Processor in a General Purpose Workstation Environment*, Ph.D. Thesis, University of Glasgow.
[20] D. Koufaty, Marr D.T. Marr, "Hyperthreading technology in the netburst microarchitecture", *IEEE Micro*, Vol. 23, No. 2, March-April 2003.
[21] S. Martello, P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, J. Wiley & Sons, 1990.
[22] D. Lammers, "Intel cancels Tejas, moves to dual-core designs", EE Times, http://www.eet.com/, May 07, 2004.
[23] P. Denning, "Working Sets Past and Present", *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 1, Jan. 1980.