HARDWARE SCHEDULING FOR DYNAMIC ADAPTABILITY USING EXTERNAL PROFILING AND HARDWARE THREADING

Brian Swahn *and* Soha Hassoun
Tufts University, Medford, MA
swahn@ece.tufts.edu
soha@cs.tufts.edu

ABSTRACT

While performance, area, and power constraints have been the driving force in designing current communication-enabled embedded systems, post-fabrication and run-time adaptability is now required. Two dominant configurable hardware platforms are processors and FPGAs. However, for compute-intensive applications, neither platform delivers the needed performance at the desired low power. The need thus arises for custom, application-specific configurable (ASC) hardware.

This paper addresses the optimization of ASC hardware. Our target application areas are multimedia and communication where an incoming packet (task) is processed independently of other packets. We innovatively utilize two concepts: *external profiling* and *hardware threading*. We utilize an M/M/c queueing model to profile task arrival patterns and show how profiling guides design decisions. We introduce the novel concept of hardware threading which allows on-the-fly borrowing of unutilized hardware, thus maximizing task-level parallelism, to either boost performance or to lower power consumption. We present a scheduling algorithm that synthesizes a hardware-threaded architecture, and discuss experimental results that illustrate adaptability to different workloads, and performance/power trade-offs.

1. INTRODUCTION

Ubiquitous, communication-enabled embedded systems appear in homes, offices, cars, military equipment, and in many other aspects of our life. While performance, area, and power constraints have been the driving force in designing many current systems, post-fabrication configurability is now needed. Configurability allows system re-usability and extends its longevity. It also permits designers to continue to debug part of the system past the expensive fabrication point, thus meeting shorter design cycles. Furthermore, configurability yields systems that can optimize energy and performance when adapting to external stimuli.

Processors and Field Programmable Gate Array (FPGA) are typical configurable platforms. The use of processors, both general purpose controllers (e.g. ARM processor[2]) as well as special purpose processors (e.g. Texas Instruments' TMS320C6000[14]), has dominated. A recent trend in configurable processors is creating *customizable* or *parameterizable* processor cores[13]. This is done by augmenting the instruction set or by specifying additional functional units and then synthesizing a custom configurable processor and its compiler, thus allowing customization at several levels: architectural, micro-architectural, and the physical/logical level.

The main advantage in using processors is software programmability. The availability of high-level programming languages and compilers allow exploiting data parallelism to produce reasonably efficient applications. The main limitation, however, is decreased performance and performance density, which can be measured as bit operations per square area[5]. As was noted by DeHon[6], this decrease in performance density is due to two main reasons:

- *underutilization*: functional units take up resources but do not continually deliver performance, and
- *over generality*: providing a hardware unit that is more complicated than needed decreases the performance density when compared to a custom unit.

FPGAs were introduced by Xilinx Corporation in 1986 and have since evolved considerably. The limitation of the initial bit-level only operations has been enhanced by adding on-chip multipliers and embedded processors. The recent Virtex-II Pro from Xilinx, for example, offers four embedded IBM PowerPC 405 processors. Such FPGAs are used heavily in industry as implementation platforms and often for rapid prototyping.

FPGAs are attractive because of the availability of CAD support and the quick design turn-around time, however, their two major limitations are low performance and high power consumption. The underlying fabric is simply too general. George et al. show that 65% of power consumption in an FPGA is associated with interconnect[8].

While configurability can be achieved by programming or customizing a processor, by FPGAs, or a combination, sometimes performance requirements under power constraints prohibit using any of these platforms. Furthermore, the need to produce high volumes justifies costly custom application-specific configurable (ASC) hardware. A number of application-specific configurable architectures have been proposed. Examples are RaPiD[7], Paleiades[15], GARP[9], and the Chameleon processor[12]. These architectures vary in their granularity (fine v.s. coarse), routing resources, in configuration abilities, and in their underlying computational model (SIMD v.s. MIMD).

A major hurdle in developing ASC hardware is the lack of general design methodologies and automated computer-aided design (CAD) tools to support exploring the configurable design space. The one commonly used technique is *internal profiling*, where a software program identifies characteristics and needs particular to an application domain. This was for example used in RaPiD [7], and more recently by Huang and Malik[10]. The authors propose to identify the computationally intensive loops in their applications. Configurable datapaths are then designed for different loops and implemented on an architectural model based on a master processor and a reconfigurable co-processor consisting of multiple functional units. We refer to this type of profiling as *internal* because it only captures the dynamics of the applications themselves (e.g. running applications from Mediabench), but not the dynamics of the external environment.

In this paper we discuss ASC optimization by innovatively utilizing two techniques: *external profiling*, and *hardware threading*. *External profiling* is key in partitioning an ASC design that is adaptable to its operating environment. External profiling has been aggressively used in designing dynamic power management schemes. In the survey by Benini et al., the authors classify such schemes as either predictive, assuming deterministic response times, or stochastic optimum control schemes capable of modeling uncertainty in the environment and the system under consideration[3]. While such models have been adopted to design dynamic power management policies, none have been used

ICCAD'03, November 11-13, 2003, San Jose, California, USA.

directly to drive hardware design or in synthesizing ASC architectures. *Hardware Threading* (HT) is a novel paradigm for creating ASC architectures. It is based on borrowing unutilized resources and then *threading* them to boost the performance or reduce power. Thus, unutilized hardware is *threaded* or *stitched* together to produce a more optimal hardware configuration. HT in essence culls resources to maximize parallelism within a task. Similar to how a processor's performance is improved by maximizing instruction-level parallelism (through super scalar architectures and advanced compiler technologies), task-level parallelism can be utilized to enhance ASC performance. We provide a high-level synthesis scheduling algorithm that results in hardware than can *dynamically* exploit task-level parallelism.

Our intended applications here are multimedia and communication application domains. These two domains require *on-demand* high-performance processing capabilities. Two distinct properties characterize such systems. First, each incoming packet can be processed *independently* of others. *Parallel* hardware processing can therefore improve performance. Second, the workloads are *dynamic* differing drastically depending on the external environment. These two characteristics allow the use of stochastic processes and queueing theory to analyze system performance and to guide the synthesis of optimal hardware-threaded architectures.

The goals of this paper are to: (a) explore using external profiling to guide custom configurable hardware design, (b) introduce the concept of HT, (c) argue its feasibility based on profiling the external environment using a formal queueing model, (d) describe a scheduling algorithm for HT, and (e) provide supporting case studies. The paper begins with a template architectural model suitable for our application domains, and then we tackle each of these goals. Examples are provided to illustrate the main concepts. We conclude with a summary and remarks about future work.

2. TEMPLATE ARCHITECTURE FOR TARGET APPLICATION DOMAINS

As mentioned earlier, we target multimedia and communication application domains characterized by the ability to process tasks[1] of the workload independently. An architectural candidate for such applications is one that utilizes independent *Processing Elements*, PEs, for compute-intensive operations, as shown in Figure 1. This architecture is well suited for our application domains because it maximizes the processing of independent parallel tasks.

The PEs are identical and custom designed to process a particular compute-intensive task or a set of tasks. For example, in a network processing application, the PEs may implement cyclic redundancy checking for ATM and Ethernet applications, and/or pseudo random generation to accelerate congestion avoidance. The PEs thus contain registers to hold incoming data from memory or intermediate results. Their interaction with memory and the outside interface however is dictated by a master processor.

The master processor monitors and schedules all activities. Upon the arrival of a task through the outside interface, the processor determines if the task should be routed to a particular PE, depending on its availability, or queued in memory. The processor implements a dynamic scheduling scheme to match tasks with PEs and to move data between memory, the PEs, and the outside interface. The processor can schedule tasks to form a long virtual pipeline among the PEs. The processor also manages the power consumption and thus shuts power for all unused PEs through clock gating. The PEs, master processor, and the memory could potentially be part of a larger SoC.

We concentrate in this paper on the optimization of the *configurable design space* among the PEs, and not on scheduling the tasks in the master processor. In Section 3, we use external profiling to determine the optimal number of PEs. In Section 4, we utilize a high-level synthesis approach to generate efficient hardware-threaded PE architectures. We apply HT to temporarily *borrow* unutilized pipeline stages from other PEs to boost the performance or to reduce the power of tasks running on active PEs. Borrowing resources involves adding interconnect and steering data among shared PEs. The interconnect allows the shared PEs to function as multiple independent PEs or as one *threaded* PE. With borrowed resources, a task can complete much faster, thus shortening the time consumed by each task. Alternatively, the frequency and V_{DD} may be reduced to save power. The effects of increased delays and interconnect capacitance due to the change in hardware must be weighed against the resulting increased adaptability.

[1]For the rest of the paper we will use the word *task* to refer to an independent packet or thread that arrives and must be executed.

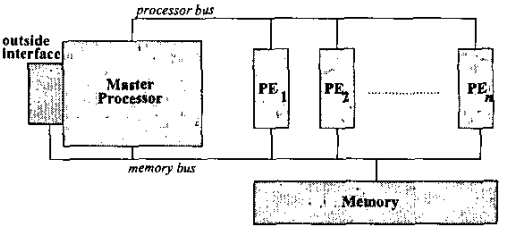

Figure 1: Template architecture for target application domains: multimedia and communication.

3. EXTERNAL PROFILING

3.1. Workload and PE Modeling

Our target applications perform on-demand execution. We can assume that the environment provides a dynamic workload. The **workload** is characterized by a mean arrival rate (λ) of tasks and by the pattern in which they arrive. We assume that the random inter-arrival times are a sequence of independent, identically distributed random variables and model them as an exponential distribution. Such a distribution has the *Markov* or *memoryless* property which states that if time t has elapsed with no arrivals, then the distribution of further waiting time is the same as it would be if no waiting time had passed. That is, the system does not remember that t time units have produced no arrivals. This is certainly the case in network processors where the arrival of packets is independent of others. We will assume that the environment's workload will continue to have a mean arrival time λ for a long enough time (order of milliseconds) for the system to achieve a steady state.

Our *service time* within each PE can also be modeled as an exponential distribution. This implies that the service time remaining to complete a customer service is independent of the service already provided. The PE service time will be referred to as W_s, and the average service rate is $1/W_s$.

Because our architectural model consists of c independent PEs, we adopt a system model based on an M/M/c queueing system, which commonly assumes random exponential inter-arrival and services times with c identical servers [1]. Note that the M/M/c queueing model could also be developed for other distributions for both the workload and the service time. For example, we might choose to model *bursty* workloads, or use a hyperexponential distribution for the service time if a large variance exists relative to the mean.

In the next subsection, we review M/M/c basics, and we then pose the problem of finding an optimal ASC architecture.

3.2. M/M/c Queueing System

Based on queueing theory, our system parameters are:

- Number of identical servers or PEs, c
- Average arrival rate of tasks from the environment, λ

- Average steady state time a task spends in the PE, W_s
- Average steady state time a task spends in the queue, W_q
- Expected steady state number of tasks queued and waiting to be serviced, L_q
- Expected steady state number of tasks being serviced, L_s

Two metrics are relevant in designing and evaluating ASC architectures. The first is ρ, the *server utilization*. It indicates how efficiently the hardware is being used. It can be calculated as:

$$\rho = \frac{\lambda \times W_s}{c} \quad (1)$$

The second and more relevant metric is W, the *average wait time in the system*, because it is inversely proportional to the system throughput. It is given by:

$$W = W_s + W_q \quad (2)$$

W_s is dependent on the PE implementation. Recall that it is an *average steady state response* which takes into account the data dependencies within each task.

W_q is more complicated to calculate. It is dependent on L_q and L_s. The latter can be determined by:

$$L_s = \lambda \times W_s \quad (3)$$

If L_s is less than c, the expected steady state wait time in the queue will be zero; however, that is not the general case. W_q can be calculated as:

$$W_q = \frac{L_q}{\lambda} \quad (4)$$

The average length of the queue, or the steady state number of tasks in the queue, is determined by:

$$L_q = \frac{\rho \times C[c, L_s]}{1 - \rho} \quad (5)$$

$C[c, L_s]$ is known as *Erlang's C formula*. Given the number of PEs, c, and the expected steady state number of tasks being serviced, L_s, Erlang's C formula specifies the probability that an arriving task must queue for PE service. It is:

$$C[c, L_s] = \frac{\frac{L_s^c}{c!}}{(1-\rho) \times \sum_{n=0}^{c-1} \frac{L_s^n}{n!} + \frac{L_s^c}{c!}} \quad (6)$$

Utilization, ρ, is restricted to be less than 1; otherwise, the average length of the queue L_q and thus W_q will be either infinite or negative. The system simply becomes overloaded.

3.3. Finding an Optimal ASC Architecture

We are interested in designing an optimal ASC architecture under different workload arrival rates λ. Given a fixed area, and several possible PE implementations that vary in their implementation area and thus their performance (W_s), we wish to choose the optimal number of *PE*s that will provide the best performance under different workload conditions.

The system model described in the previous subsections can be used to find the optimal number of independent PEs to use for a set of known arrival times that characterize the outside environment. The designer of an ASC architecture for a specific application will specify a mean arrival rate of tasks entering the system, λ. The expected steady-state time a job spends in the system, W, can then be computed from W_s, which is implementation specific, and W_q from the previous equations.

Problem: Given a set of arrival times $\Lambda = \{\lambda_1, \lambda_2, ...\lambda_k\}$, a fixed area A, and several possible implementations for a PE, find c, the number of PEs, that maximize throughput across all given $\lambda \in \Lambda$.
Solution Our problem can be solved as follows. Because the total area is fixed and we know the area for each possible PE implementation, we can determine all possible cs. For each $\lambda_i \in \Lambda$, we can then calculate W using the equations presented in Section 3.1. Our task is then to choose the best c that minimizes the overall system wait time W. We illustrate this with an example.
Example: We are given a total area of 100 square units, three different PE designs (PE_a, PE_b, PE_c), each that has different area and service time trade-offs as shown in Table 1. We first determine $c = \lfloor \frac{A}{A_i} \rfloor$, where A_i is the implementation area for PE design i. The results are shown on the last row of the table.

	PE_a	PE_b	PE_c
Service time in PE, W_s	10	8	6
PE area, A_i	10	15	25
# of PEs, c	10	6	4

Table 1: Three different PE designs that trade performance for area. Given a fixed area of 100 units square, we calculate the number of possible PEs for each design.

We can then compute W for arrival rates $\Lambda = \{0.3, 0.4, 0.5, 0.6, 0.7\}$. Figure 2 shows the service wait time in the queue and in the PEs for the different arrival rates for all three designs. Because there are 10 PE_as, and each PE_a has less resources than in PE_b and PE_c, the W_s for PE_a is the longest. However, W_q is the smallest and remains small across the examined λs. In contrast, W_q is more prominent for designs PE_b and PE_c once the arrival rate exceeds $\lambda = 0.6$. The queueing time for PE_c at $\lambda = 0.7$ is infinite because the utilization exceeds 1. If we know a priori that our arrival rates are within certain ranges and the likelihood of each arrival rate then we can certainly choose an optimal number of *PE*s. For example, if $\lambda < 0.6$, then PE_c would be the PE implementation of choice. After carefully choosing a PE implementation that minimizes W_q across all λs, hardware threading can be used to *thread* the collection of PEs together and create a dynamically adaptable system.

4. HARDWARE THREADING

4.1. Concept and Synthesis Flow

Hardware threading is the ability to borrow unused resources from idle PEs and thread them together to maximize task-level parallelism. PEs can then operate either independently without threading, i.e. as a non-threaded PE, or with n-threading when resources from n PEs are collected to create a *threaded* PE. The master processor in Figure 1 sets the configuration mode based on a scheduling policy that monitors task arrival rates, queueing activities, task service times within the PEs, and PE availability. Such a policy must be designed to maximally take advantage of the adaptable n-threaded hardware capabilities. Our focus here is providing CAD tools and a synthesis flow that enable designing threaded PEs.

Figure 3 shows an overview of the HT synthesis flow. The PE functionality is specified using a control data flow graph (CDFG) that can contain conditionals and loops. A state-based schedule where each state corresponds to a pipeline stage can be generated using ASAP scheduling utilizing a technique similar to the path-based scheduling (PBS) technique[4]. Area or resource constraints guide the scheduling algorithm. Next, the ASAP schedule is processed by our HTScheduler described in Section 4.3. HTScheduler produces an HT schedule that guides the synthesis of an n-threaded PE. The resource constraints here are those assigned to n PEs. HTScheduler traverses the states in the ASAP schedule trying to look ahead by one or more states to schedule future operations

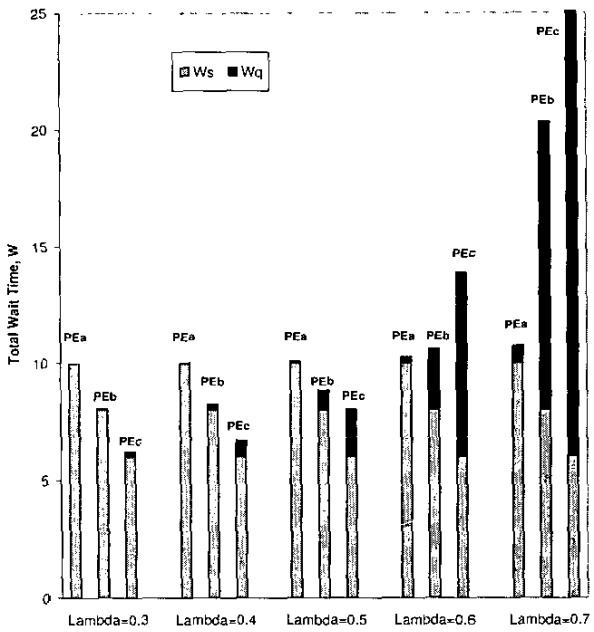

Figure 2: Average total wait time in the system, W, which is the sum of W_q and W_s, for three PE designs at different arrival rates.

earlier thus maximizing the usage of all available resources. The number of future states examined is referred to as the *window size*. Finally, the HT schedule is translated to RTL-level HDL which can then by synthesized using a tool like Design Compiler from Synopsys. Before describing the details of the algorithm, we provide a detailed example.

4.2. Example

The following code implements a function f.

```
int f(a,b,c,d,e,f)
{
    var1 = a + b;           // +1
    var2 = c + d;           // +2
    var3 = var1 + var2;     // +3
    var4 = e + f;           // +4
    if(var3 < var4)         // <1
       var5 = var3 * var4;  // *1
    else {
       var6 = var1 * var2;  // *2
       var7 = var2 * var3;  // *3
       var5 = var6 + var7;  // +5
    }
    return (var5);          // :=1
}
```

Figure 4(a) illustrates the corresponding CDFG. Each operation in f has a label. For example, label +1 refers to the first add operation. The vertices in the CDFG correspond to operations; the edges represent data and control dependencies.

Our resource constraints are: one adder and one multiplier are available in one clock cycle. An add and a multiply operation can be scheduled at the same time only if they do not have any data dependencies. The corresponding ASAP schedule is shown in Figure 4(b). The resulting schedule has 9 states or pipeline stages. Physical registers are added to implement these pipeline stages and they are sometimes added to provide temporary storage of intermediate operands as needed. The number and locations of these registers become finalized after logical and physical synthesis where retiming will relocate the registers to optimize frequency and area.

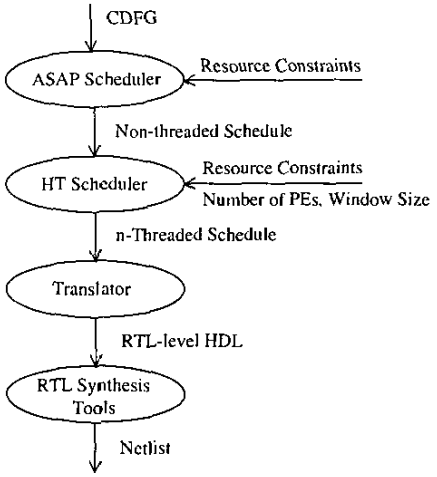

Figure 3: Synthesis flow to produce hardware-threaded architectures.

The HTScheduler uses the ASAP schedule to produce a new HT schedule. The threading of two PEs is conceptually shown in Figure 4(c,d), with Figure 4(d) showing the effective threaded schedule. The effective threaded schedule is used when the PEs are operated in a threaded mode. With two PEs, two adders and two multipliers are available for scheduling in one clock cycle. Thus, operations +1 and +2 can be scheduled simultaneously, as well as operations +3 and +4 in the absence of data or control dependencies among each pair. The dashed ellipses in Figure 4(c) indicate operations that will be combined into a single state, or a super state. The solid edges are the original sequencing edges from the ASAP schedule. The dashed ones indicate changes in the next state from the ones in the original schedule. For example, the dashed edge from +3 to < 1 indicate that state < 1 becomes the next state after super state (+3,+4), skipping state +4 in-between.

When implemented, the operations within a super state must multiplex their inputs to reflect that sometimes the inputs come from the predecessor state (no threading), or from a neighboring PE, or the outputs of an earlier state. The multiplexing essentially allows on-the-fly adaptation.

The resulting 2-threaded schedule is outlined by the new states and the dashed edges. The threaded version reduces the number of states along the worst case path to 6. The reduction in states directly corresponds to a reduction in task execution time, and results in an overall lower service time for the PEs. The master processor can thus choose to schedule a task on either a single PE (a PE operating in a non-threaded mode) or a threaded PE, depending on the task arrival rates and queueing activities. If the arrival rates are low, then certainly having the threading option would allow to improve the overall service time without affecting the queue wait time, W_q. However, if the arrival rates are high, then the master processor will choose to maximize resource utilization for all PEs and thus schedule tasks on single non-threaded PEs. Failure to do so would cause an unnecessary increase in queue wait time. Also note that the master processor's task scheduling policy can benefit from user specified directives that specify the task arrival rates, if known, and/or the desire to maximize performance or power savings.

4.3. HT Scheduling Algorithm

Our scheduling algorithm preserves all control and data dependencies in the non-threaded ASAP schedule while creating the new threaded implementation. The algorithm is shown in Figure 5. The inputs are: a non-threaded ASAP schedule, the relevant PE resources used when creating the ASAP schedule, the number of

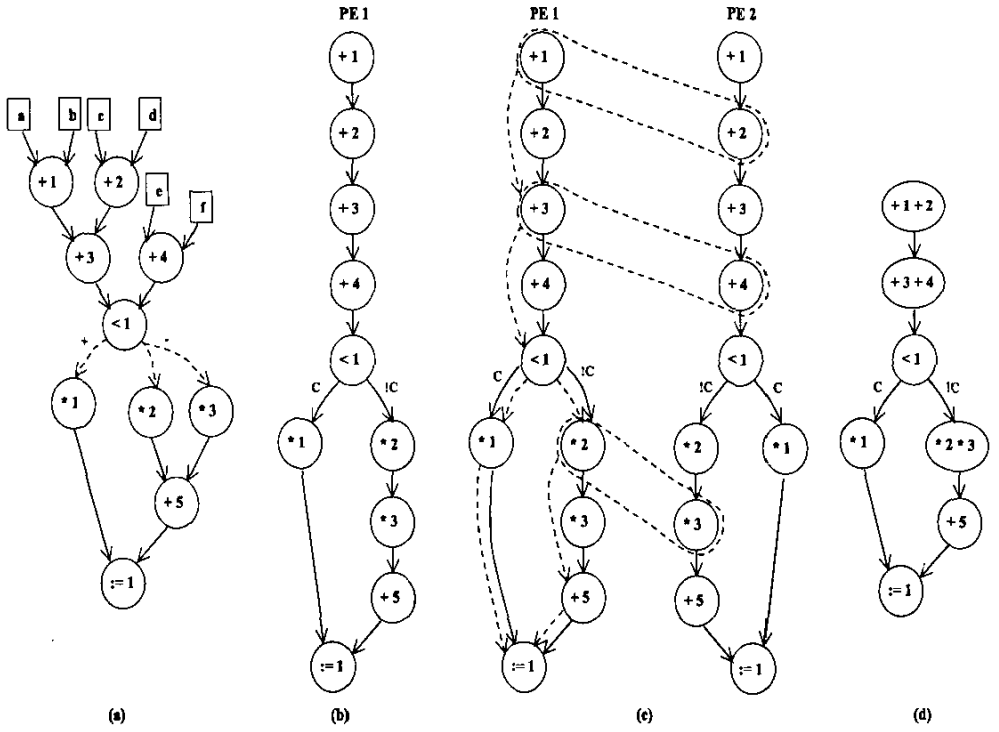

Figure 4: Example to illustrate HT concept and synthesis process. (a) CDFG. (b) ASAP schedule. (c) 2-Threaded schedule (d) Conceptual HT schedule

PEs, and a window size that specifies the number of downstream states that will be examined while traversing the ASAP schedule.

The following variables are used. *CurrentState* represents a state in the non-threaded schedule. The scheduler's goal is to schedule more operations within this state. *NextState* is a state in the non-threaded schedule which is being considered for scheduling with *CurrentState*. If *NextState* is schedulable with *CurrentState*, the two states are scheduled together and the states are said to be threaded together. *ByPassedEdges* is a queue which holds information about edges in the non-threaded schedule. These edges have their sources already scheduled and their sinks potentially form an entry point for a new edge. That is, these edges will eventually become bypassed once new edges are inserted. The *Borrowed* variable is a queue which holds all the states that have been combined with the current state. *StatesAlreadyScheduled* is an array which holds information about all states that have been already scheduled. The resulting threaded schedule is a modified version of the non-threaded schedule where some states become super states with new operations, and new edges connect the super states to the proper next states.

The *CurrentState* is initially set to the first state in the non-threaded schedule (see line 1). All states in the non-threaded schedule are traversed until the end of the schedule is reached (2, 28). If the *CurrentState* is a conditional state, the scheduler is recursively called with a subschedule containing all the states along each conditional path until a join state or a terminal state is reached (3-5). Once a state is examined (6), it is marked as already scheduled (7) and the next states within a window of size w are examined (8-18).

The IsSchedulable routine (14) checks resource constraints, and for control and data dependencies between the *CurrentState* and *NextState*. If the *NextState* can be scheduled with the *CurrentState*, then *NextState* as well as its edge are added to the respective queues. The *NextState* is marked as borrowed and operations of *CurrentState* and *NextState* are combined (18).

Once all of the states in the window are examined, the edges for the threaded schedule are created (19,27). The new edge may be created between the *CurrentState* and the successor state (27), or one or more states is skipped due to combining operations within the skipped states with states earlier in the schedule (20-25). In the latter case, the edges and states are skipped one at a time until all *Borrowed* states are skipped.

Upon completion, the resulting schedule is a modified version of the non-threaded schedule complete with edges for threaded and non-threaded modes of operation. The hardware for the PEs can be derived from this schedule with both modes of operation inherent.

The run time complexity of our algorithm is $O(s^2)$, where s is the number of states in the original ASAP schedule. The algorithm essentially visits each node in the ASAP schedule and in the worst case examines a window size of s. However, as we demonstrate using our case studies, it is often that a small window size (up to 3) is sufficient to cull operation level parallelism in common applications. Thus, assuming a small constant window size, the algorithm's run time is linear in the number of states.

We have implemented this algorithm in C++. We also implemented the HT schedule translation into synthesizable Verilog. The case studies in the next section apply our design flow to generate the threaded architectures. We also explain how using the different modes of operation allow trading performance for power.

5. CASE STUDIES

We used in our case studies the following set of resource constraints to generate our ASAP schedule: one adder, one multiplier, two comparators, one divider, two incrementers, a cosine lookup, and an exponential lookup. A 2-threaded architecture used twice these resources to generate the optimal 2-threaded architecture.

```
HTscheduler (S,R,n,w)
Input:  S ≡ Non-threaded Schedule
        R ≡ Resource Constraints Per PE
        n ≡ Number of PEs
        w ≡ Window Size
Vars:   CurrentState ≡ Current State to be Scheduled
        NextState ≡ Next State to be Scheduled
        ByPassedEdges ≡ Queue of Edges to Bypass
        Borrowed ≡ Queue of Borrowed States
        StatesAlreadyScheduled ≡ Array of States Already Scheduled
SideEffect:Modify S to Become the Hardware – Threaded Schedule
1.  CurrentState = initialState(S);
2.  while CurrentState is not Null do
3.    if IsConditional(CurrentState) then
4.      for each child of CurrentState
5.        HTscheduler(Schedule(child),R,n,w)
6.    else if StatesAlreadyScheduled(CurrentState) = 0 then
7.        StatesAlreadyScheduled(CurrentState) = 1
8.        NextState = CurrentState
9.        for ( i = 0; i ≤ w; i++ ) do
10.         NextState = successor(NextState)
11.         if IsConditional(NextState) ∨ IsJoining(NextState) then
12.           i = w
13.         else
14.           if IsSchedulable(CurrentState,NextState,R,n) then
15.             ByPassedEdges.append(outedge(NextState))
16.             Borrowed.append(NextState)
17.             StatesAlreadyScheduled(NextState) = 1
18.             CombineStates(CurrentState,NextState)
19.   if successor(CurrentState) = Borrowed.top then
20.     Borrowed.dequeuetop
21.     while !IsEmpty(Borrowed)∧
            Borrowed.top = (target(ByPassedEdges.top)) do
22.       Borrowed.dequeuetop
23.       ByPassedEdges.dequeuetop
24.     CreateEdge(CurrentState,target(ByPassedEdges.top),dashed)
25.     ByPassedEdges.dequeuetop
26.   else
27.     CreateEdge(CurrentState,successor(CurrentState),dashed)
28.   CurrentState = successor(CurrentState)
```

Figure 5: HT scheduling algorithm.

We followed the synthesis flow outlined in Figure 3. Our final threaded schedule was translated to Verilog HDL. We then used Synopsys' Design Compiler to generate a netlist that implements our threaded architectures. For our case studies, we report the area and clock periods obtained from Design Compiler reports. All threaded architecture results ($w > 0$) included the overheads associated with HT, such as: area, timing, capacitive loads, etc.

5.1. Detailed Case Study: DCT

The Discrete Cosine Transform (DCT) is widely used in image compression techniques such as JPEG and MPEG. Our synthesis flow outlined in Figure 3 was followed to generate DCT's HT architecture. We first generated DCT's CDFG and non-threaded ASAP schedule. The non-threaded schedule was used as an input to our scheduling algorithm and a corresponding threaded schedule was generated with window sizes ($w = 1, 2, 3, 4$). The resulting threaded schedules were transformed into Verilog HDL and synthesized using Synopsys' Design Compiler.

The results are shown in Table 2 under the group column heading "DCT". The column with a window size of zero represents the non-threaded schedule. The columns show the optimization results for a window sizes 1 and ≥ 2. The number of states in the original schedule is 14. The schedule with the least states (12 states) was achieved with a window size of 2. All operation-level parallelism was exploited with a window size of 2; as window sizes beyond 2 did not further reduce the number of states in the schedule. The next two rows report the area and clock period for each schedule. The normalized areas and normalized clock periods are listed for window sizes 1 and ≥ 2. The area overhead is due to additional multiplexors needed to select between the inputs to some of the states. The clock period increase was caused by additional delays due to: (a) the introduction of multiplexors along the critical paths, and (b) the additional interconnect delay and increased fan out load on some signals. In order to properly model these loads, our designs were synthesized under a wire load model, with a load slope of 0.311, where the slope was used to estimate the loads associated with the wire lengths.

The service times for the non-threaded and threaded schedules were simulated and compared using an abstract cycle-based Verilog model. The system consisted of tasks arriving independent of one another and we recorded the number of cycles spent in the queue (Wq) and processing time (Ws). The total time was obtained by multiplying the cycle count times the clock periods we obtained after hardware synthesis. As the arrival rate changed, the system wait times varied as shown in Figure 6. For larger λs the non-threaded ($w = 0$) architecture had the best service time, however, the threaded architecture provided a more efficient solution at smaller λs. The service time for $w = 1$ shows a 6.2% improvement over the service time for $w = 0$ at $\lambda = 1.9E-4$. Similarly, for $w = 2$, a 12.4% improvement in performance was possible. This data is shown in Table 2 in the row entitled Performance Increase. It should be noted, while the execution time per task on each threaded PE is reduced, the overall system wait time doesn't gain as much due to the reduction of the number of available PEs.

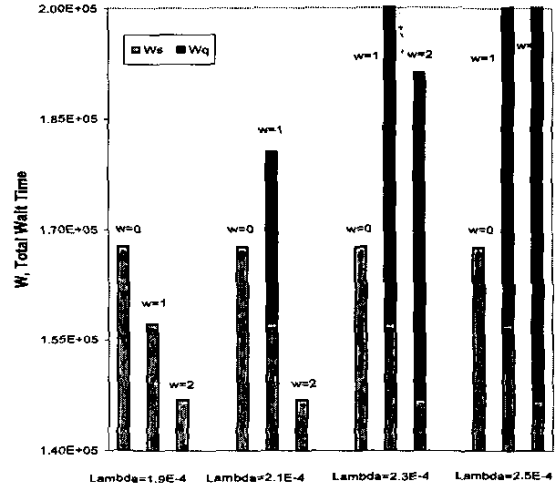

Figure 6: Total wait time for the DCT case study

A threaded mode of operation can be used to lower average system power while maintaining a constant system wait time similar to that of a non-threaded mode of operation. Given that $P_{dyn} = C_L V_{DD}^2 f$, power can be reduced by decreasing the supply voltage and clock frequency.

We compare the power consumption of the non-threaded and 2-threaded architecture that was synthesized using a window size of two. By allowing the threaded architecture's Ws to be equivalent to that of the non-threaded architecture's, the frequency of the threaded architecture can be scaled down. For example, the lowest λ in Figure 6 shows a $Ws = 1.67E6$ for the non-threaded architecture and $Ws = 1.46E6$ for the threaded architecture, given $Ws = T \times \#clock\ cycles$. By setting $Ws_{threaded} = Ws_{non-threaded}$ and using the number of threaded clock cycles, the reduction in frequency can be calculated. In this example, the frequency can be reduced by 13% such that the new low power frequency is 87% of the old normal power frequency ($f_{new} = 0.87 f_{old}$).

The largest power reduction is achieved not only by decreasing the frequency, but also by scaling the supply voltage. In this example, the supply voltage can be scaled down since the frequency is reduced. This allows the normalized delay to be increased. The normalized delay here is $1/f$, which equates to 1.14. If the origi-

	DCT			DFS		DFT			FFT	
Window Size	0	1	≥ 2	0	≥ 1	0	1	≥ 2	0	≥ 1
# of Schedule States	14	13	12	9	8	10	9	8	13	12
Area[1]	6585	1.053	1.081	5412	1.013	6890	1.001	1.014	9068	1.129
Clock Period [2]	15.99	1.015	1.033	16.98	1.026	15.54	1.062	1.155	16.50	1.029
Performance Increase[3]	-	6.2%	12.4%	-	6.6%	-	2.5%	3.3%	-	4.3%
Power Reduction w/ f and V_{DD} scaling[4]	-	20%	44%	-	19%	-	8%	9%	-	13%

Table 2: Number of schedule states, area, timing, performance improvements, and power reduction estimates for case studies. [1]The area for a window size of zero is reported in square units; the area for window sizes of one and two are normalized to the area in the zero-window size case. [2]The clock period for a window size of zero is reported in nanoseconds; the clock period for window sizes of one and two are normalized to the clock period in the zero-window size case. [3]Best possible increase with a negligible Wq. [4]Power reduction achieved by scaling f and V_{DD}.

nal supply voltage was 5V, then the supply voltage can be reduced to ~3.75V, given a normalized delay curve[11]. The voltage and frequency scaling for this case study shows $P_{new} = 0.56P_{old}$. The threaded architecture allows a substantial savings in power for the same system wait time as the non-threaded architecture. This result is reported in the row entitled Power Reduction w/ f and V_{DD} scaling in Table 2.

5.2. Other Cases

We applied our synthesis procedure and analysis to a few signal processing examples, including: the Discrete Fourier Series (DFS), the Discrete Fourier Transform (DFT), and the Fast Fourier Transform (FFT). The results are shown in Table 2. Over all four case studies, it is shown that 3%-12% improvement in performance are possible when using hardware threading. Combining hardware threading with frequency and voltage scaling, it is possible to obtain power savings in the range of $8\% - 44\%$.

6. CONCLUSION

This paper addressed using external profiling and hardware threading to synthesize and optimize dynamically adaptable application-specific architectures. While external profiling is not a new concept, to our knowledge, this the first work in high-level synthesis that addresses the synthesis of dynamic datapaths based on *dynamic* workloads, or using *external profiling*. The novelty of our Hardware Threading technique lies in dynamically exploiting unutilized resources to maximize task-level parallelism. This is a very strong and useful concept in any type of hardware configurability. Our case studies show $3\% - 12\%$ improvement in performance, and a range of $8\% - 44\%$ savings in power.

7. REFERENCES

[1] A. Allen. *"Probability, Statistics, and Queueing Theory with Computer Science Applications"*. Academic Press, Inc., 1990.

[2] ARM. "http://www.arm.com/armtech/ARM11Microarchitecture".

[3] L. Benini, A. Bogliolo, and G. De Micheli. "A Survey of Design Techniques for System-Level Dynamic Power Management". *IEEE Transactions on VLSI Systems*, 8(3):813–33, August 2000.

[4] R. Camposano. "Path-Based Scheduling for Synthesis". *IEEE Transactions on Computer-Aided Design*, 10(1):85–93, January 1990.

[5] A. DeHon. "Reconfigurable Architectures for General-Purpose Computing". *AI Technical Report 1596, MIT AI Lab, Cambridge, MA.*, 1996.

[6] A. DeHon. "The Density Advantage of Configurable Computing". *IEEE Transactions on Computers*, 33(4):41–49, September 2000.

[7] C. Ebeling, D. Cronquist, and P. Franklin. "RaPiD - Reconfigurable Pipelined Datapath". In *International Workshop on Field-Programmable Logic and App lications*, 1996.

[8] V. George, H. Zhang, and J. Rabaey. "The Design of a Low Energy FPGA". In *ISPLED*, pages 188–93, 1999.

[9] J. R. Hauser and J. Wawrzynek. "Garp: A MIPS Processor with a Reconfigurable Coprocessor". In *Proc. of the International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 24–33, 1997.

[10] Z. Huang and S. Malik. "Exploiting Operation Level Parallelism Through Dynamically Reconfigurable Datapaths". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, 2002.

[11] J. Rabaey. *"Digital Integrated Circuits"*. Prentice Hall, 1996.

[12] B. Salefski and L. Caglar. "Re-configurable computing in wireless". In *Proc. of the ACM/IEEE Design Automation Conference (DAC)*, pages 178–83, 2001.

[13] Tensilica, Inc. "http://www.tensilica.com/".

[14] Texas Instruments, Inc. "http://www.ti.com/".

[15] M. Wan, H. Zhang, V. George, M. Benes, A. Abnous, V. Prabhu, and J. Rabaey. "Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System". *IEEE Journal of VLSI Signal Processing*, 2000.