Synthesis of Reconfigurable High-Performance Multicore Systems

Jason Cong   Karthik Gururaj   Guoling Han
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA, 90065

ABSTRACT

Reconfigurable high-performance computing systems (RHPC) have been attracting more and more attention over the past few years. RHPC systems are a promising solution for accelerating system performance, lowering power consumption and minimizing operation cost. In order to achieve high performance on this hybrid system, it is important to effectively explore the design space, which includes accelerator synthesis, resource allocation and job scheduling. In this paper we propose novel algorithms for reconfigurable resource allocation and job scheduling to optimize performance of multicore RHPC systems. Specifically, we first propose an interesting approximation algorithm to assign jobs to processors with consideration of coprocessors at the global optimization step. Then we present an optimal solution for coprocessor selection in the local optimization step. In this paper we also demonstrate that designers can quickly explore a large number of accelerator design choices with the help of high-level synthesis tools. Experiments show that our proposed techniques provide efficient solutions for real-life benchmarks and generate higher quality of results. When compared to other heuristic algorithms, our results can achieve up to 47% performance improvement.

INTRODUCTION

The relentless pace of Moore's law has led us to the era of multicore microprocessors. A number of microprocessor manufactures, such as Intel, IBM, AMD and Sun MicroSystems, have launched their due-core and quad-core processor products. We expect to see an increasing number of cores with each advance in feature size. However, modern microprocessors are approaching performance limits by simply increasing the clock frequency. Wire delay, design variation, power consumption and heat dissipation make it more and more difficult to improve the performance of single threaded applications by putting more transistors on chip [1]. However, to process the massive amounts of data in the "Era of Tera" [7], we will see an increase in the demand for computing. Then, the challenging question is how to boost system performance steadily for future computations.

High-performance computing based on integrating general-purpose multicore microprocessors and field-programmable gate arrays (FPGAs) has been attracting increased interest over the past few years. First, customization of FPGAs for each application allows optimized application-specific memory structures, massive parallelism and deep pipelining. In addition, FPGA technology still has abundant growth potential in capacity and clock rate. Thus, an application-specific coprocessor can more likely achieve better performance than the conventional general-purpose CPUs. Second, the heat dissipation and power consumption of FPGAs are significantly less than general-purpose CPUs. Leveraging configurable coprocessors can dramatically reduce the system power consumption and lower the operation costs of data centers. Finally, the reconfigurability of FPGAs provides sufficient flexibility to migrate computation between the hardware and software in order to adapt to a wide range of applications and evolving specifications and standards.

A number of high-performance computing vendors, such as Cray [26], SGI [33] and SRC [34], have offered their architectures to integrate microprocessors and reconfigurable fabrics for the high-end server market. XtremeData [35] and Altera [25] have also provided FPGA coprocessor solutions for AMD Opteron-based systems. These evolving hybrid systems have presented a "plug and play" architecture solution to developers for adding processing power without modifying their infrastructure. Moreover, high bandwidth connections, such as the HyperTransport bus from AMD and Front Side Bus from Intel, have significantly reduced the communication time between core processors and FPGAs, which was a major bottleneck in traditional reconfigurable computing systems. On the software side, high-level synthesis tools, such as AutoPilot [24], Catapult [27], and Impulse C [28] have made programming reconfigurable devices much easier. Developers can still work on the high-level C or C-like languages, and synthesis tools can automatically compile the code down to hardware implementations. A number of recent research efforts [6][18] have successfully applied high-level synthesis tools for applications in the high-performance computing domain. The advances in software support, as well as architectures, promote adoption of the FPGA-based acceleration.

A crucial step to achieving high performance in this hybrid system is to effectively explore the huge design space, which includes accelerator synthesis, resource allocation and job scheduling. Over the last two decades, there have been tremendous efforts that involved solving problems related to reconfigurable computing. A comprehensive survey of reconfigurable computing can be found in [4][12][15]. [2][9][10] present system architectures and compilation tools, which use reconfigurable resources to implement coprocessors for single applications. Design space exploration for coprocessor synthesis has been addressed in [20][21][14][3][17][23]. Scheduling problems on parallel machines have been extensively studied as well in the past decades. The complexity, heuristic and approximation algorithms have been discussed in [11][8][22][19][13]. However, little attention has been paid to the synthesis and job scheduling

problem for multicore RHPC systems. Under the new scenario, the heterogeneous multiprocessor systems are used to perform a number of diverse computation-intensive applications, such as financial analytics, physical simulation and data mining. Most of these are routinely executed as batch jobs on multiprocessor or multicore systems. How to use reconfigurable resources effectively for the jobs and schedule them on processors is a key to maximizing the system performance and lowering costs; this has motivated us to investigate the problem. In our work, we especially focus on the system performance optimization problem.

The contributions in this work are that we present a systematic synthesis flow and novel algorithms for system performance optimization under an area constraint for multicore RHPC systems. Specifically, in the global optimization stage we propose a 2-approximation algorithm to assign jobs to processors which guarantees to generate a solution with objective function value of at most two times the optimum. In the following local optimization step, we propose a dynamic programming algorithm to finally select the coprocessor configurations to maximize system performance. We prove that the problem can be optimally solved. We also discuss the techniques used to achieve design tradeoffs for coprocessor implementations. We demonstrate that the tradeoffs can be conveniently obtained with the help of high-level synthesis tools.

The remainder of the paper is organized as follows. We first present our problem formulation in Section 0. The synthesis flow and algorithmic details for solving the performance optimization problem under area constraint are described in Section 0, followed by experimental results in Section 0. Finally, conclusion remarks are presented in Section 0.

PROBLEM FORMULATION

In this work we are targeting the reconfigurable high-performance computing systems which consist of multicore processors and FPGA fabrics that are connected by high-bandwidth connections such as AMD's HyperTransport or Intel's Front Side Bus (as shown in Figure 1). The cores in the multicore processor can be either heterogeneous or homogeneous. The reconfigurable resources are shared among all the cores and can be used to implement coprocessors for the most computation-intensive kernels. Applications can access the coprocessors through well-defined APIs managed at the OS layer. The APIs define the protocol between CPUs and FPGA coprocessors, and transfer data and control information to/from the coprocessors. Applications need to transfer data to the local memory resident in the FPGAs before coprocessors start to execute. Calculated results will be fetched from the out-buffer of the coprocessors to the microprocessors' memory.

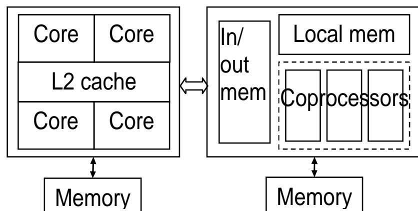


Figure 1. Block diagram of a multicore coupled with FPGA accelerators

The programs running on the system are a set of independent computationally demanding jobs. For example, stock option pricing calculation, pattern analysis, spectral analysis, and similarity detection are routinely executed jobs on servers in a finance institute. In our work, we assume that every job is to be processed by exactly one processor. Job j takes p_{ij} time units when processed by processor i, which can be obtained by profiling tools.

To accelerate these jobs, computation-intensive kernels can be identified and become candidates for coprocessor implementation. Considering a job with software execution time t_{sw}, a portion of the job can be implemented on a coprocessor on FPGAs. The execution time of the coprocessor is t_{hw}, and the communication time spent to transfer data between core processors and FPGAs is t_{comm}. Then we can reduce the execution time by $t_{sw} - t_{hw} + t_{comm}$. In our work, the FPGA resources are statically allocated to a set of jobs executed regularly on servers. Once configured, the same set of jobs will run over a long period of time or indefinitely. Hence, FPGA configuration time is negligible.

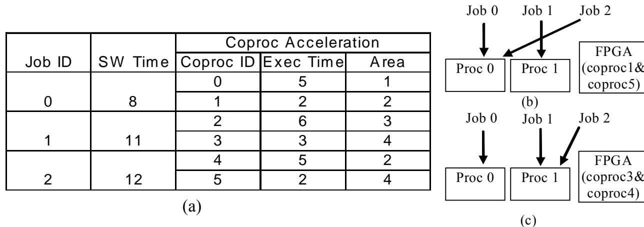
Usually, we may have a number of implementation choices for a coprocessor to make tradeoff between performance and cost. Suppose a coprocessor of job j has multiple implementation choices. A particular implementation choice k for job j can reduce the execution time of job j on processor i by Δp_{ijk} with area $c_{jk;}$ we will use a tuple $<\Delta p_{ijk}, c_{jk}>$ to represent the configuration in our paper. Since we have limited FPGA area, it is important to allocate the FPGA resources appropriately to the coprocessors so that the system performance can be maximized. In addition to the FPGA resource allocation, job assignment is equally important in order to achieve the optimal performance since we have multiple cores in the system. In a multiprocessor system processing batch jobs, one of the most important optimization criterions is *makespan*, the maximum processing time on the processors. Therefore, we formulate our resource-constrained RHPC (RC-RHPC) optimization problem as follows.

RC-RHPC synthesis problem: Given n microprocessors, m independent jobs and their profiling information, the coprocessor configuration candidates for each job, and area budget C for FPGA chips, select a subset of coprocessor configurations to be implemented on FPGAs, and schedule the tasks on the n microprocessors so that the makespan is minimized under the area constraint.

Theorem 1. The decision problem of RC-RHPC synthesis is NP-complete.

Proof: A general multiprocessor scheduling problem can be reduced to a RC-RHPC problem by setting the area budget to 0. Since the multiprocessor scheduling problem is NP-complete [8], the RC-RHPC synthesis problem is also NP-complete.□

Note that this problem is different from the scheduling and binding problems in the traditional high-level synthesis (HLS). In HLS, each operation has a fixed execution time whereas each job in our problem may have a large number of coprocessor configurations which lead to various execution times. In addition, resource constrained scheduling problem assume that the number and type of resources are given before scheduling. However, in our problem, we need to consider both coprocessor selection and scheduling to maximize performance under a total area constraint.



Job ID	SW Time	Coproc Acceleration		
		Coproc ID	Exec Time	Area
0	8	0	5	1
		1	2	2
1	11	2	6	3
		3	3	4
2	12	4	5	2
		5	2	4

(a)

Figure 2. An example of RC-RHPC problem

Figure 2(a) shows an example of three tasks to be processed by two identical microprocessors. The total FPGA area is six units. The numbers in the second column show the software processing time p_{ij} for each task. In this example, each application has only one coprocessor, and each coprocessor has two possible implementations. As mentioned before, we need to solve coprocessor selection and job assignment in the RC-RHPC problem. A simple approach, that we call greedy algorithm, is to solve the two subproblems separately. In this approach, the first step is to select coprocessors so that the maximum execution time reduction can be achieved. Then coprocessors 1 and 5 will be selected. After that, the minimum makespan we can obtain is 11 as shown in Figure 2(b). However, considering coprocessor selection and job assignment simultaneously can lead to the solution shown in Figure 2(c) with a makespan of 8. For this example, the inferior makespan is about 37.5% longer than the optimal one. It indicates that the two subproblems depend on each other and need to be carefully considered at the same time in order to achieve the optimal makespan.

PROPOSED SYNTHESIS FLOW AND ALGORITHMS

We are proposing the three-step approach as shown in Figure 3 to minimize makespan under the area constraint. The first step is to explore the design space and generate all the coprocessor design points and obtain their performance/cost tradeoff curves. Then, we perform coprocessor-aware job scheduling, which assigns jobs to processors with consideration of coprocessor acceleration and cost. After the job assignment is done, the last step selects the coprocessor configurations to achieve the minimal makespan. We shall discuss the three steps in detail in the following subsections.

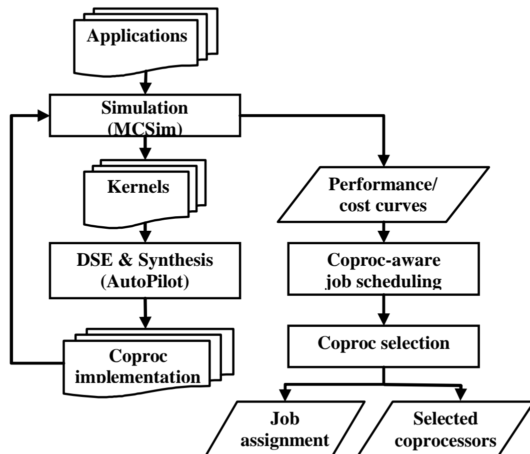


Figure 3. Proposed RHPC synthesis flow

1.1 Design Tradeoff Generation

When designing a hardware accelerator, architects are usually able to tune a number of parameters to achieve different performance and cost. However, the design exploration process is time consuming and error-prone if designers are working at the register transfer level. Therefore, usually only a small portion of the design space can be explored by manual design. With the advance of high-level synthesis tools, the exploration process becomes much easier. Designers can reconstruct the specification at behavioral level. In addition, most high-level synthesis tools can perform program transformation and optimizations with the directions from designers. AutoPilot [24], the commercial synthesis tool used in our experiments, provides synthesis options and pragmas so that different constraints and optimization techniques can be applied to the same specification to obtain various design points. In general, there are many transformation and synthesis techniques which can be applied to obtain design tradeoffs. In our experiments, we have selected the following techniques to demonstrate that design space can be explored at a higher abstraction level by using high-level synthesis tools. Next, we shall briefly discuss the techniques used in this work for design tradeoff exploration.

Resource constraint: If parallelism is abundant in a design, simply setting resource constraints (i.e., the number of adders, multipliers, etc.) can lead to various design tradeoffs. Increasing resources can let more operations be executed in parallel, thus shortening application latency. When resources are increased to a certain number, the performance cannot be further improved due to the available instruction-level parallelism.

Module selection: The current FPGA vendors provide IPs for certain operations. For example, the Xilinx Floating-Point Core [31] offers designers a parameterized library of floating point operation implementations. These cores can be customized to allow optimization for latency, frequency or area. Module selection is the process of selecting specific modules for each operator to meet design constraints.

Loop pipelining: Loop pipelining [16] is an optimization technique to realize temporal parallelism by scheduling different iterations to be executed in an overlapped fashion. The initiation interval, the time interval between consecutive executions of its steady state, depends on the resource constraint as follows. $II \geq \max_{r \in resources}(uses(r)/number(r))$ where $number(r)$ and $uses(r)$ are the number of allocated resources and the number of uses of resource r in one loop iteration. Therefore, different implementations can be generated by setting initiation intervals.

Loop parallelization: We use loop parallelization to refer any optimizations that take advantage of the loop-level parallelism. This can include loop fusion, loop splitting, loop unrolling, loop distribution, etc., and all combinations.

1.2 Coprocessor-Aware Job Assignment

In this step, we are going to assign jobs to processors and take the generated coprocessors into consideration at the same time. We observe that this can be modeled as an extension of the generalized assignment problem.

Generalized assignment problem: Schedule n independent jobs onto m unrelated parallel machines. Job j takes p_{ij} time when processed by machine i, and incurs a cost c_{ij}, i=1,…,m, j=1,…,n. Given values C and T, we wish to decide if there exists a schedule of total cost at most C such that the makespan is at most T.

The generalized assignment problem can be mathematically formulated as an integer linear programming problem $ILP(T,C)$ shown above. In this formulation, we introduce a 0-1 integer variable x_{ij} for each machine-job pair. x_{ij} is 1 if job j is assigned to processor i. Equation (1) and (3) ensure that the total cost and makespan are no more than C and T respectively. Equation (2) indicates that each job should be processed by one and only one processor.

$$LP(T,C): \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} \le C \quad (1)$$

$$\sum_{i=1}^{m} x_{ij} = 1 \quad \text{for j} = 1,...,\text{n} \quad (2)$$

$$\sum_{j=1}^{n} p_{ij}x_{ij} \le T \quad \text{for i} = 1,...,\text{m} \quad (3)$$

$$x_{ij} \ge 0 \quad \text{for i} = 1,..., \text{m, j} = 1,...,\text{n} \quad (4)$$

$$x_{ij} = 0 \quad \text{if } \text{p}_{\text{ij}} > T \quad \text{for i} = 1,..., \text{m, j} = 1,...,\text{n} \quad (5)$$

The best approximation algorithm so far is proposed by Shmoys and Tardos [19], which gives a polynomial-time 2-approximation algorithm. The algorithm is based on solving linear relaxations of $ILP(T,C)$, denoted as $LP(T,C)$, and then rounding the fractional solution intelligently to a nearby integer solution. Specifically to obtain the integer solution, a bipartite graph is constructed followed by a minimum-cost matching. The details can be found in the paper [19]. Based on the proof in that paper, we can have the following theorem.

Theorem 2. If $LP(T,C)$ has a feasible solution, then there exists a schedule that has makespan of at most $2T$ and cost no more than C.

We shall discuss the extension of the generalized assignment problem in the remainder of this section. In our problem, if a coprocessor configuration k is used, job j can be finished in time $p_{ijk}=p_{ij}-\Delta p_{ijk}$ on machine i with incurred cost c_{jk}. For each job j, a coprocessor implementation k' is dominated by implementation k if $\Delta p_{ijk'} \le \Delta p_{ijk}$ and $c_{jk'} \ge c_{jk}$. The dominated implementations can be pruned out to get a pareto optimal set. Thus, the pareto frontier can be plotted on a graph with time along the X-axis and cost along the Y-axis. We can define a convex pareto set as an ordered subset of the design configurations such that the piecewise linear interpolant f connecting each configuration, taking the execution time as the independent variable, is a convex function. The value of f increases as the processing time decreases. Two adjacent configurations define an interval where the linear interpolant l_i for this interval is a straight line connecting the two configurations. The slope of l_i increases monotonically which indicates that the reduction in execution time increases at a decreasing rate with the amount of resource employed. The concept has also been used in [1] to efficiently perform system evaluation. Figure 4 shows a convex pareto set for a design.

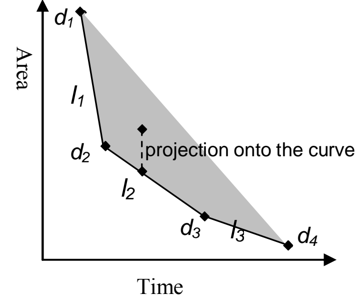


Figure 4. Convex pareto set

Without loss of generality, we can assume that w potential configurations are available for each job. Given the w design points d_1, d_2, …, d_w, a linear combination of these points can be represented as $x_1d_1 + ... + x_wd_w$, where the real numbers $x_i \ge 0$ and their sum equals 1. The set of all linear combinations constitutes a convex hull as shown in the shaded area of Figure 4.

Our job assignment algorithm is based on the approximation that assumes all the design points in the convex hull are available. Then we can formulate the job assignment problem as the following mixed integer linear program $LP_EXT(T,C)$, where x_{ij} is a 0-1 integer variable and x_{ijh} is a real variable. In this program, x_{ij} is 1 if job j is assigned to processor i. x_{ijh} is the parameter used to determine a design point in the convex hull defined above. Note that for a given i and j, the sum of all the x_{ijh} (for h=1,…,w) equals 1 if x_{ij} is 1. In addition, the p_{ij} and c_{ij} are the execution time and cost for a design point in the convex hull.

$$LP_EXT(T,C): \; p_{ij} = \sum_{h=1}^{w} x_{ijh}p_{ijh} \quad \text{for i} = 1,..., \text{m, j} = 1,...,\text{n}$$

$$c_{ij} = \sum_{h=1}^{w} x_{ijh}c_{ijh} \quad \text{for i} = 1,..., \text{m, j} = 1,...,\text{n}$$

$$x_{ij} = \sum_{h=1}^{w} x_{ijh} \quad \text{for i} = 1,..., \text{m, j} = 1,...,\text{n}$$

$$\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij} \le C$$

$$\sum_{i=1}^{m} x_{ij} = 1 \quad \text{for j} = 1,...,\text{n}$$

$$\sum_{j=1}^{n} p_{ij} \le T \quad \text{for i} = 1,...,\text{m}$$

$$x_{ijh} \ge 0 \quad \text{for h=1,..., w, i} = 1,..., \text{m, j} = 1,...,\text{n}$$

$$x_{ijh} = 0 \quad \text{if } \text{p}_{ijh} > T \quad \text{for h=1,..., w, i} = 1,..., \text{m, j} = 1,...,\text{n}$$

Due to the complexity of mixed integer programs, it is practically infeasible to obtain exact solutions for the LP_EXT problem with large sizes. Interestingly, we find that the following approximation, a linear relaxation followed by careful rounding, is a 2-approximation algorithm.

Theorem 3. If the linear relaxation of $LP_EXT(T,C)$ has a feasible solution, then there exists a schedule with makespan of at most $2T$ and cost at most C.

Proof: The theorem can be proven by constructing a schedule from the fractional solution of the linear program. For each machine-job pair, x_{ij} represents the portion of job j that will be executed on machine i in the linear program solution. If $x_{ij}>0$, the corresponding processing time on machine i is $p'_{ij}=p_{ij}/x_{ij}$ with associated cost $c'_{ij}=c_{ij}/x_{ij}$. For the machine-job pairs where x_{ij} is equal to 0, we set $p'_{ij}=+\infty$ and $c'_{ij}=0$.

Using T, C, p'_{ij} and c'_{ij} as parameters, we can construct a generalized assignment problem instance $LP(T,C)$. Interestingly, the vector x is a solution to the constructed $LP(T,C)$. Based on Theorem 2, we can round x to integers so that the makespan of the rounded solution is at most $2T$ in the $LP(T,C)$. By using the rounding solution to the $LP_EXT(T,C)$ problem, we find that $p'_{ij}x_{ij}=p_{ij}$ and $c'_{ij}x_{ij}=c_{ij}$ if x_{ij} is rounded to 1. Otherwise, $p'_{ij}x_{ij} = p_{ij} = 0$ and $c'_{ij}x_{ij} = c_{ij} = 0$.

Since $\sum_{j=1}^{n} p_{ij} = \sum_{j=1}^{n} p_{ij}x_{ij} \le 2T$ and $\sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij} = \sum_{i=1}^{m}\sum_{j=1}^{n} c'_{ij}x_{ij} \le C$ in the rounded solution, we can see that the makespan is at most $2T$ and cost at most C in the $LP_EXT(T,C)$□

Actually, we can refine the approximation by assuming that only the design points along the convex curve are available. Since the piecewise linear function is convex, we can guarantee that the point is in the convex hull, as illustrated in Figure 4. Projecting the points obtained from $LP_EXT(T,C)$ vertically onto the curves, we obtain a legal solution without an increase to the total cost and makespan. In fact, we can prove that the design points will always be on the curve if a tight cost constraint is given.

1.3 Coprocessor Selection

The previous step assigned jobs to processors by approximating their execution times and costs using convex piecewise linear functions. It assumes that all the design points can be obtained along the linear curves; however, only a number of discrete design points may be available. Therefore, this coprocessor selection step is to select the coprocessor configurations under the area and job assignment constraints so that the makespan is minimized.

Given a value T, the corresponding decision problem is whether we can select a subset of coprocessor configurations so that the system makespan is no more than T while the total area used is no more than C. If we can solve the decision problem, the minimal makespan can be obtained by a binary search of T and solving a series of decision problems.

Let S_i denote the set of tasks mapped onto processor i. CL_i is the list of coprocessor implementation choices for the tasks in S_i. If $MinA_i(T)$ is the minimal area requirement for processor i so that its total execution time is no more than T, we can prove the following lemma.

Lemma 1. Makespan T is feasible *iff* $\sum_{i=1}^{m} MinA_i(T) \le C$.

Based on this observation, we only need to calculate $MinA_i$ for i=1,…, m, which can be solved by dynamic programming. Suppose T_i is the total execution time of S_i without any coprocessor acceleration, $MinA_i$ is 0 if T_i is no more than T. Otherwise, let $\Delta T_i=T_i-T$, that is, ΔT_i is the time needed to be saved by accelerators. If CL_i is empty, we cannot shorten the makespan by using coprocessors. Thus, we set the $MinA_i$ to $+\infty$. Based on the solutions of the subproblems, we can define the recurrence function $g(CL_i, \Delta T_i)$ for computing $MinA_i$ as:

$$g(CL_i, \Delta T_i) = \begin{cases} 0 & \text{if } \Delta T_i \le 0 \\ +\infty & \text{if } CL_i \text{ is empty} \\ \min(g(CL_i', \Delta T_i), g(CL_i', \Delta T_i - \Delta p_{ijk}) + c_{jk}) & \text{otherwise} \end{cases}$$

where CL'_i is the remaining coprocessors after the first coprocessor configuration in the list, whose tuple is $\langle \Delta p_{ijk}, c_{jk} \rangle$, has been eliminated from consideration. Note that we may have multiple implementation choices for a particular coprocessor. We need to eliminate all the other choices if an implementation of a coprocessor has been used. The problem has $O(\Delta T_i N)$ complexity, where N is the total number of coprocessor configurations.

Since the decision problem can be optimally solved, we can use a binary search to find the minimal makespan under the given job assignment. Therefore, we can draw the following conclusion:

Theorem 4. The coprocessor selection problem can be optimally solved in pseudo polynomial time.

EXPERIMENTAL RESULTS

We have implemented our algorithm in a C++/Unix environment. LPsolve [30], a publicly available linear programming solver has been used to solve linear programs. In this experiment, the target FPGA device is Xilinx Virtex4 FPGA.

We selected the following ten applications from finance computation [29], scientific computation, lithography simulation, cryptography, bioinformatics and media processing as the jobs to be accelerated as shown in Table 1.

Table 1. Benchmarks

Benchmark	# Configuration
Monte Carlo Black-Scholes simulation	9
Currency option	9
Johnson approximation in stock option calculation	5
FFT	10
Lithography simulation	5
Random number generation	8
Triple des encryption / decryption	3
Smith waterman	1
Canny edge detection	1
MPEG4 decoder	2

First, we modified these programs to make them synthesizable. Then applying the tradeoff generation techniques discussed in Section 1.1, we manually tweaked the C programs and synthesized a number of design implementations for each computation-intensive kernels. Table 1 lists the number of coprocessor implementation choices for each application.

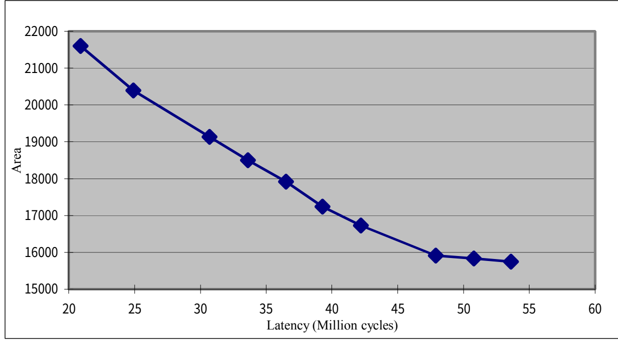


Figure 5. Design points for a FFT coprocessor

For example, Figure 5 shows the FFT coprocessor implementations generated by AutoPilot on Xilinx Virtex4 FPGAs. The FFT example is used in spectral transform shallow water model simulation and consists of single precision arithmetic operations. In its kernel computation, we have nested loops which are frequently executed. In coprocessor implementations, the loops can be pipelined to achieve better performance. As discussed in the previous section, if setting smaller initiation interval in loop pipelining, we can get better throughput and use more resources. For example, when II is 2, the coprocessor uses 21547 slices. When we relax II to 8, the used slices reduce to 16726 slices. In the meantime, we can generate various customized floating point IP cores using Xilinx floating point library. For example, Table 2 lists different implementations for floating pointer add/sub IP. When latency increases, the core uses for slices, however can run at a higher frequency. We can select different configurations to generate various implementations.

Table 2. Different implementations for fadd/sub

Latency	#Slices	Frequency(MHz)
2	280	96
4	293	141
6	318	140
8	398	244
10	407	307
12	429	335

In Figure 5, the area is the total number of used slices. The design with the minimum resource usage saves area by about 29% compared to the design with the largest area. However, its performance is 2.56X slower than the fastest implementation.

Similarly, Figure 6 shows another design tradeoff curve for a hardware implementation (on FPGA) of the lithography imaging simulation [6]. The implementation in [6] follows a polygon based approach and uses fixed-point arithmetic for maximizing performance on FPGA. The kernel of the application consists of a set of nested for loops; the iterations of the innermost loop can be performed in parallel. Loop unrolling and pipelining can be used to improve the performance of the application. In [6], the authors propose a novel data partitioning technique to overcome the memory bandwidth problem for on-chip RAM blocks in the FPGA. Several coprocessor implementations can be obtained using different data partitioning sizes (Figure 6). Increasing the number of partitions allows a larger number of processing elements to be used. This leads to lower latencies at the cost of higher area consumption on the FPGA.

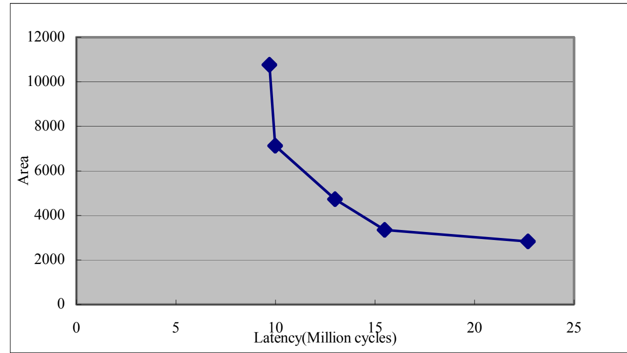


Figure 6. Design points for a litho simulation coprocessor

For each application, we list the number of possible implementation choices in the second column of Table 1. The set of implementation choices provides designers with enough flexibility to choose the best implementation for their design.

1.4 Simulation Methodology

In order to accurately measure the performance of the hybrid system, we have developed a multicore simulation framework, namely *MC-Sim*, which uses SESC [32] as our core simulator. SESC models a full out-of-order pipeline with branch prediction, caches, and every other component of a modern processor necessary for accurate simulation. A hardware coprocessor's performance can be precisely measured by simulating the design at register-transfer level. However, the extremely slow simulation speed makes it only applicable for small designs or stimulus. Therefore, the conventional method cannot be used in our architecture exploration. In our work, we built a tool which generates cycle-accurate ANSI C performance models for coprocessors after a high-level synthesis process. The generated models can simulate state transitions based on inputs and calculate latency accurately. Through well-defined interfaces, the C performance models can be accessed as a library call in *MC-Sim*. Besides the models for core processors and coprocessors, a cycle-accurate communication model is developed as well which models the communication delay for transferring data between FPGAs and core processors in the system. The details of the simulator can be found in [5]. We compared the accuracy of our generated C performance models with the corresponding RTL SystemC models generated by AutoPilot. The experimental results show that the C models have the same accuracy while running at least 50X faster than RTL models for small-sized designs. For large designs, the speedup is even much higher.

1.5 RHPC Synthesis Results

In our experiment we model a system consisting of multiple 6-issue, out-of-order cores running at 2GHz. Each core has a separate 64KB L1 instruction and data cache. A 1MB L2 cache is shared among all the cores. Note that the coprocessors latency will be scaled in *MC-Sim* by a factor that is the ratio between the frequency of general-purpose cores and coprocessors. We applied our algorithm to various FPGA area constraints. Figure 7 shows the final system makespan with the given area budgets. System makespan decreases as more FPGA area is given. For a 2-core

system, the makespan with 10K slices is 2.67X larger than the one with 100K slices. When more cores are added to the system, the trend still holds. However, the speedup is not as significant as the 2-core system.

Figure 8 shows the final FPGA area usage under various budget constraints. Since our algorithm guarantees that the area usage will not be larger than the given area budget, we can see that the curves are always under the area budget line.

To evaluate the efficiency of the proposed algorithm, we also compare it to the aforementioned greedy approach that solves the scheduling and resource allocation independently. Figure 9 shows the percentage of makespan reduction by using our proposed method compared to the greedy one. We can see that our proposed method almost always outperforms the greedy algorithm. When the area budget increases, our proposed algorithm shows more advantage when considering coprocessors and scheduling simultaneously. On average, our algorithm can reduce makespan by 29%, 36% and 47% for 2-core, 4-core and 8-core systems respectively.

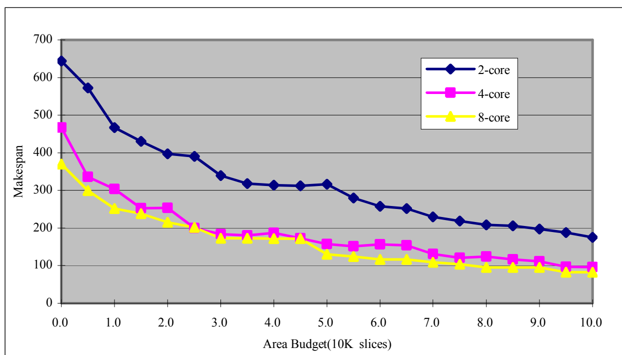


Figure 7. System performance under various area budgets

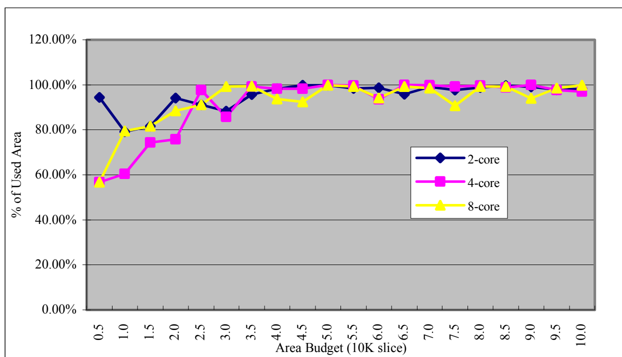


Figure 8. Actual area usage

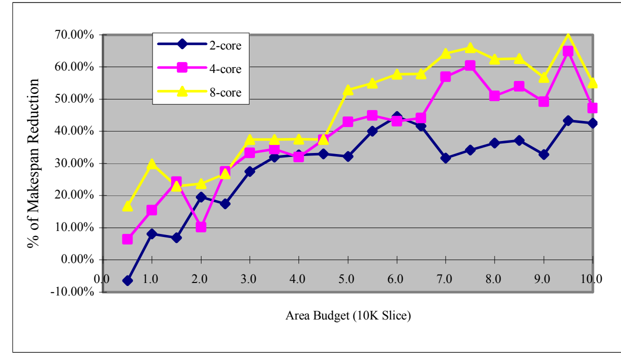


Figure 9. Comparison of the two algorithms

Finally we measured the execution time of our solver as shown in Table 3. We used all the ten benchmarks and synthesized for 2-core, 4-core and 8-core systems respectively. We list the number of variables and constraints generated using our LP formulation. The last row shows the total execution time when the program running at a 2.6GHz Xeon CPU with 1GB memory. Since our proposed algorithm only needs to solve LP programs, the execution time is fast. And we can also apply our scalable algorithm to larger problems.

Table 3. Execution time

configuration	2-core	4-core	8-core
# variables	40	80	160
# constraints	58	104	196
execution time (sec)	0.39	0.62	1.5

CONCLUSION

Reconfigurable high-performance multicore systems have been attracting more and more attention over the past few years. In this paper we present efficient algorithms for reconfigurable resource allocation and job scheduling for this hybrid system. We also present techniques for coprocessor design tradeoff generation, and demonstrate that designers can quickly explore a large number of design choices with the help of high-level synthesis tools. The experimental results show that our algorithms can effectively utilize reconfigurable resources to shorten system makespan, resulting in quality that is up to 47% better than a simple heuristic algorithm.

REFERENCES

[1] S.G. Abraham, B.R. Rau and R. Schreiber, "Fast Design Space Exploration Through Validity and Quality Filtering of Subsystem Designs", *HP Labs Technical Reports*, HPL-2000-98.

[2] T.J. Callahan, J.R. Hauser and J. Wawrzynek, "The Garp Architecture and C Compiler," *Computer*, vol. 33, No. 4, pp.62-69, 2000.

[3] S. Chaudhuri, S. A. Blythe, and R. A. Walker, "A Solution Methodology for Exact Design Space Exploration in a Three-Dimensional Design Space," *IEEE Transaction on Very Large Scale Integration Systems,* vol. 5, issue 1, pp. 69-81, 1997.

[4] K. Compton, S. Hauk, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Computing Surveys*, vol. 34, pp. 171-210, 2002.

[5] J. Cong, K. Gururaj, G. Han, A. Kaplan, M. Naik, and G. Reinman, "MC-Sim: An Efficient Simulation Tool for MPSoC Designs," International Conference on Computer-aided Design, 2008.

[6] J. Cong and Y. Zou, "Lithographic Aerial Image Simulation with FPGA-Based Hardware Acceleration," *in Proc. of 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2008.

[7] P. Dubey, "Recognition, Mining and Synthesis Moves Computers to the Era of Tera," *Technology@Intel Magazine*, Feb. 2005.

[8] M.R. Garey and D.S. Johnson, "Strong NP-Completeness Results: Motivation, Examples and Implications," *Journal of Assoc. Comput.*, Mach. 25, pp. 499-508, 1978.

[9] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer*, vol. 33, No. 4, pp. 70-77, 2002.

[10] M.B. Gokhale, J.M. Stone, "NAPA C: Compiling for a Hybrid RISC/FPGA Architecture," in *Proc. of the IEEE Symposium for Custom Computing Machines*, pp. 126-135, 1998.

[11] R.L. Graham, "Bounds for Certain Multiprocessing Anomalies," *Bell System Tech. J. 45*, pp. 1563-1581, 1966.

[12] S. Hauk, A. Dehon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*, Morgan Kaufmann, 2007.

[13] K. Jansen, M. Mastrolilli, "Approximation Schemes for Parallel Machine Scheduling Problem with Controllable Processing Times," *Computers and Operations Research*, Vol. 31, pp. 1565-1581, 2004.

[14] V. Kathail, S. Aditya, R. Schreiber, B. Rau, D. Cronquist, and M. Sivaraman, "PICO: Automatically Designing Custom Computers," In *IEEE Computer*, pp. 39-47, Sept. 2002.

[15] M. Gokhale and Paul S. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*, Springer, 2005.

[16] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *Proc. 1998 Conference on Programming Language design and Implementation*, pp. 318-328, 1988.

[17] M. Palesi and T. Givargis, "Multi-Objective Design Space Exploration Using Genetic Algorithms," In *Proc. of the Tenth International Symposium on Hardware/Software Codesign*, pp. 67-72, 2002.

[18] A.R. Putnam, D. Bennett, et al., "CHiMPS: a High-Level Compilation Flow for Hybrid CPU-FPGA Architectures," *in Proc. of 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2008.

[19] D.B. Shmoys and E. Tardos, "An approximation algorithm for the generalized assignment problem," *Mathematical Programming*, vol. 63, pp. 461-474, 1993.

[20] B. So, P.C. Diniz, M.W. Hall, "Using Estimates from Behavioral Synthesis Tools in Compiler Directed Design Space Exploration," in *Proc. of the 2003 Design Automation Conference*, pp. 514-519, 2003.

[21] B. So, M. Hall, and P.C. Diniz, "A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems," in *Proc. of the 2002 Conf. on Programming Languages Design and Implementation,* pp. 165-176, June 2002.

[22] R. Vickson, "Two Single Machine Sequencing Problems Involving Controllable Job Processing Times," *AIIE Transactions*, vol. 12, pp. 258-262, 1980.

[23] G. Wang, W. Gong, B. DeRenzi, R. Kastner, "Design Space Exploration Using Time and Resource Duality with the Ant Colony Optimization," in *Proc. of the 43rd annual conference on Design Automation*, pp. 451-454, 2006.

[24] AutoPilot, http://www.autoesl.com

[25] Altera Corp., http://www.altera.com.

[26] Cray Inc., http://www.cray.com

[27] Catapult, http:://www.mentor.com

[28] Impulse C, http://www.impulsec.com

[29] Finance Numerical Recipes, http://finance.bi.no/~bernt/gcc_prog/recipes/recipes/

[30] LPsolve, http://www.cs.sunysb.edu/~algorith/implement/lpsolve/implement.shtml

[31] Xilinx Inc., http://www.xilinx.com.

[32] SESC simulator, http://sesc.sourceforge.net

[33] SGI, http://www.sgi.com

[34] SRC Inc., http://www.srccomp.com

[35] XtremeData, Inc., http://www.xtremedata.com