The CQSA Toolkit: A Tool for Effective and Efficient Slack Allocation for NoC-based MPSoCs

Deliverable Report, Task ID 1795.001

Task Leader: Professor Donald E. Thomas
Brett H. Meyer, Adam S. Hartman, and Donald E. Thomas Carnegie Mellon University
Department of Electrical and Computer Engineering
Pittsburgh, PA 15213
{bhm, ahartman, thomas}@ece.cmu.edu

November 30, 2009

Abstract

Wear-out related permanent faults are projected to make system lifetime a critical issue for all designs. In embedded systems, lifetime can be increased using *slack*, underutilization in execution and storage resources, so that when components fail, data and tasks can be re-mapped and re-scheduled. The design space of possible slack allocation is both large and complex. However, based on the observation that useful slack is often *quantized*, we have developed an approach, *Critical Quantity Slack Allocation* (CQSA), that effectively and efficiently allocates execution and storage slack to jointly optimize system lifetime and cost. In this report, we briefly summarize the motivation for and organization of CQSA, and then detail how to install and use CQSA to find the lifetime-cost Pareto-optimal slack allocations for a simple example.

I. Introduction

Wear-out related permanent faults are projected to make system lifetime a critical issue for all designs. Recent research has shown that when integrated circuit wear-out failure mechanisms are not compensated for, system failure rates increase 365% when scaling from 180 to 65 nm, and exponentially beyond [1]. This lifetime degradation is in large part due to various parameters, including current density, V_{Th}, and V_{DD}, not scaling ideally as features shrink. This leads to higher system temperatures, worse sub-threshold leakage, and thus, faster device breakdown. As a result, all systems must be designed with lifetime in mind.

Our research is focused on design-time computer system architecture optimization for embedded network-on-chip multiprocessor systems-on-chip (NoC-based MPSoCs). When a processor or switch fails in an NoC-based MPSoC, repair is impossible.[1] However, the system may continue to operate if the design includes sufficient *slack*—execution and storage resources beyond those required in the initial configuration. We assume that systems can automatically detect component failure (for example, using [2]), at which point they reboot, re-mapping tasks and data from failed resources and to those with slack and re-routing traffic. If performance constraints are still satisfied, system failure is averted [3].

Given a performance- and cost-constrained application and a fixed (NoC-based) communication architecture, our objective is to select where in the system to allocate slack, and how much to allocate, such that system lifetime and system manufacturing cost are jointly optimized. The design space of possible distributions of slack for a single communication architecture is large and complex: the number of possible slack allocations is exponential in the number of resources in the system and alternatives in the component library, and evaluating any single slack allocation requires repeated system-level performance, power, and temperature modeling. In one example we have considered, an MPEG-4 decoder with 21 processors, 5 memories and 10 switches, there are 1.6 billion possible slack allocations alone and each system lifetime evaluation took from 46.4 to 144.5 seconds [4]. Exhaustively exploring such a design space would take over 2000 years on a single workstation.

To address the complexity of slack allocation, we have developed *Critical Quantity Slack Allocation* (CQSA), a novel, scalable, generalizable execution and storage slack allocation technique [4][5]. CQSA takes advantage of the fact that the extra slack required to survive component failure is often *quantized*; we call these amounts of slack *critical quantities*. CQSA then uses the critical quantities for the network switches in the communication architecture of the NoC to efficiently focus the search for the best possible lifetime-cost trade-offs. Since the complexity of CQSA is closely tied to the number of critical quantities evaluated, and therefore the number of switches in the NoC, the search grows sub-linearly with respect the number of components in the whole system.

In this report, we detail how to go about (a) installing and (b) using the CQSA toolkit, including the use of comparison approaches we have implemented for the purpose of evaluating CQSA. For a detailed description of how the CQSA algorithm and the supporting modeling and estimation approaches work, please see Brett H. Meyer's dissertation, *Cost-effective Lifetime and Yield Optimization for NoC-based MPSoCs* [6], and related publications [4][5].

[1]We conservatively assume that memories don't experience permanent failure, as row- and column-redundancy or error-correcting codes may be inexpensively employed to address lost cells.

II. Background

At the system level, the lifetime of NoC-based MPSoCs can be improved at design-time by:

1) allocating *execution slack* by replacing low-performance processors with higher-performance processors,
2) allocating *storage slack* by replacing low-capacity memories with higher-capacity memories, or
3) changing the communications architecture by adding or modifying switches and links, and possibly adding spare processors and memories.

The task of system-level design-for-resilience is to determine how to use these strategies to cost-effectively extend lifetime.

Slack allocation on a fixed communication architecture (1 and 2) is an extraordinarily complex problem. Given a candidate communication architecture, the slack allocation design space contains up to n^m slack allocations for a system with n components and m possible alternatives in the component library. Since slack allocation is so complex and furthermore is needed to perform communication architecture exploration (3), our research has focused on slack allocation.

Slack allocations that are optimal for lifetime distribute just enough slack at just the right locations so as to maximize lifetime in the presence of permanent component failure while minimizing the increase in manufacturing cost (area). Given an application and communication architecture, our goal is to find the set of allocations of execution and storage slack that jointly optimize system lifetime and system cost.

A. *Critical Quantities*

The design challenge of slack allocation is efficiently finding the most cost-effective allocations of execution and storage slack. We have observed that the slack necessary for a system to survive component failure is *quantized*. For example, for a system to survive processor failure, enough slack must be allocated so all of its tasks can be re-mapped. Allocating less slack than is required to re-map each of the processor's tasks serves no purpose. For example, for x MIPS of execution slack to improve lifetime, there must be a component in the system that uses $y \leq x$ MIPS. If less than y MIPS of execution slack is allocated, system lifetime is not improved: not all of the component's tasks can be re-mapped, and its failure results in system failure. Furthermore, system lifetime and yield are likely *degraded*, lifetime by the increase in system temperature that follows the increase in power consumption of the upgraded processor, yield by the increase in component area and the resulting increase in vulnerability to defect. We therefore define the *critical quantity* $\langle es, ss \rangle$ of a component as the total slack, es MIPS of execution slack and ss KB of storage slack, required to re-schedule and re-map the tasks and data that would be orphaned if that component were to fail.

In the simplest case, the critical quantity of a processor with a capacity of x MIPS is $\langle x, 0 \rangle$. Likewise, the critical quantity of a memory with a capacity of y KB is $\langle 0, y \rangle$. However, a component's critical quantity is not always simply the capacity of that component, as when one component fails and a number of other components might suddenly become inaccessible. The critical quantity for a switch is the sum of the capacities of the processors and memories immediately attached to it.

B. *Critical Quantity Slack Allocation*

We therefore present *Critical Quantity Slack Allocation* (CQSA). CQSA assumes as inputs:

- a description of a performance-constrained application, including computation, storage and communication requirements for each software task;
- a fixed communication architecture for a single-chip multiprocessor, including an initial selection of processors, memories, switches and their interconnection; and,
- an initial task-resource mapping, including an assignment of computational tasks to processors, storage tasks to memories, and communication to links and switches.

In this context, CQSA jointly optimizes system lifetime and cost by determining (a) how much slack should be allocated in the system, and (b) where in the system it should be allocated, such that the system mean-time-to-failure (MTTF) is increased in the most area-efficient ways possible. System MTTF is a function of:

- the target application and given communication architecture;
- component utilization, which is a function of the tasks mapped to a given component;
- component power, which is derived from component utilization and other parameters; and,
- component temperature, which is derived from system-level temperature modeling.

System cost (area) is determined using system-level floorplanning. The result of CQSA is a set of MTTF-area Pareto-optimal designs with variable trade-offs from which the designer may select the design point(s) most appropriate for the given target application.

To achieve this goal, CQSA performs a series of design space explorations, allocating slack and evaluating the resulting cost and lifetime of the system. CQSA allocates slack by replacing low capacity processors and memories with higher capacity slack or memories, creating opportunity for failures to be survivable by enabling tasks to be re-mapped and traffic re-routed in ways that potentially still satisfy performance constraints. Design space exploration focuses on those quantities of slack

Algorithm 1 Critical Quantity Slack Allocation

```
1: // Stage 0
2: $es = 0$
3: while $es < min(cqExecutionList)$ do
4:     $sys = allocateExecSlackGreedily(sys)$
5:     $es = executionSlack(sys)$
6:
7: // Stage 1
8: for all $\langle es, 0\rangle$ in $cqExecutionList$ do
9:     $sys = allocateExecSlackExhaustively(es)$
10:    while $es \leq es_{max}$ do
11:        $sys = allocateExecSlackGreedily(sys)$
12:        $es = executionSlack(sys)$
13:
14: // Stage 2
15: for all $\langle es, ss\rangle$ in $cqExecutionStorageList$ do
16:    $sys = allocateExecStorageSlackExhaustively(es, ss)$
17:    while $es \leq es_{max}$ and $ss \leq ss_{max}$ do
18:        $esAllocation = allocateExecSlackGreedily(sys)$
19:        $ssAllocation = allocateStorageSlackGreedily(sys)$
20:        $sys = compareTradeOff(esAllocation, ssAllocation)$
21:        $es = executionSlack(sys)$
22:        $ss = storageSlack(sys)$
```

expected to result in cost-effective lifetime improvement, those defined by *critical quantities* of slack, in particular critical quantities for network switches. A detailed analysis of the lifetime improvement that is possible when focusing exploration on switch critical quantities can be found in Chapter 5 of *Cost-effective Lifetime and Yield Optimization for NoC-based MpSoCs* [6].

Using the m unique critical quantities $\{\langle es_1, ss_1\rangle, ..., \langle es_m, ss_m\rangle\}$ defined by the n system switches as starting points, $m \leq n$, CQSA performs a series of exhaustive and greedy execution and storage slack allocations.[2] By focusing exploration in this way, CQSA efficiently exposes those slack allocations that cost-effectively maximize the number of survivable combinations of processor and switch failures, while pruning away the overwhelming majority of the design space.

The CQSA algorithm is outlined in Algorithm 1 and is composed of three stages that consider different sets of critical quantities. *Stage 0* starts with the baseline architecture and incrementally allocates execution slack to find the best execution slack allocations not covered by a switch critical quantity. If a system has no switch critical quantities (e.g., if no switch failure is survivable under any circumstances), only *Stage 0* is performed.

Switch critical quantities of execution slack alone (e.g., $\langle es, 0\rangle$) go in *cqExecutionList* and are processed in *Stage 1*. Switch critical quantities of execution and storage slack (including $\langle 0, ss\rangle$) go in *cqExecutionStorageList* and are processed in *Stage 2*. Search based on a switch critical quantity proceeds in two steps: first, an exhaustive search is conducted for the allocation of the critical quantity of slack that maximizes system MTTF; second, a greedy search proceeds which incrementally allocates slack. In Stage 1, this greedy search allocates only execution slack. In Stage 2, this greedy search allocates both execution and storage slack.

A detailed walkthrough of the CQSA algorithm can be found in Chapter 3 of *Cost-effective Lifetime and Yield Optimization for NoC-based MPSoCs* [6]. More details on the system lifetime evaluation approach can be found in Chapter 4.

III. Installing CQSA

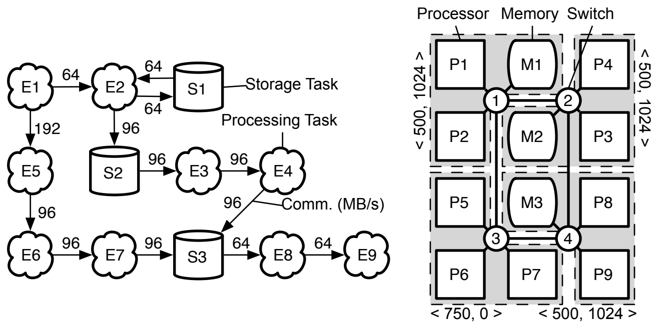
The CQSA toolkit is available for download at http://www.ece.cmu.edu/ thomas/SRC/cqsa.tgz.

A. System Requirements

To use CQSA you will need a linux or unix workstation or equivalent. We have compiled CQSA with `gcc` 3.4.6 and `make` 3.81 on SuSE Linux 9.3, as well as Mac OS X 10.4 and 10.5. Though we have not tested CQSA with a wide variety of other platforms, we do not anticipate significant portability problems.

The only other software package CQSA requires is the Gnu Scientific Library (GSL). The GSL is freely available at http://www.gnu.org/software/gsl/.

[2]We evaluate $m \leq n$ critical quantities, and not n, because some switches may have the same critical quantity. CQSA is deterministic, and any two searches that are based on the same critical quantity finds the same set of slack allocations.



(a) (b)

Fig. 1: An example (a) task graph and (b) architecture with two unique switch critical quantities, $\langle 750, 0\rangle$ for Switch 3 and $\langle 500, 1024\rangle$ for Switches 1-2 and 4.

B. Installation Instructions

To install CQSA:

1) First untar the CQSA distribution cqsa.tgz, with `tar xzvf cqsa.tgz`. This creates the `cqsa/` directory and a variety of subdirectories.
2) Next, `cd cqsa/` and open `Makefile.defs.template` in your favorite editor.
3) Uncomment (remove the `#`) either line 2 or 3 if your operating system is OS X (Darwin) or Linux respectively, and save the file as `Makefile.defs`. This is used to indicate to the rest of the distribution the location of GSL.[3]
4) Build CQSA and its supporting libraries using `make`. This compiles all the necessary libraries and related executables.

C. CQSA Distribution

The core of the CQSA algorithm and supporting system modeling algorithms (e.g., lifetime evaluation) have been written in C++. The CQSA distribution includes:

- `benchmark/` – several example benchmarks;
- `exhaustive_ra/` – source for an executable that takes any slack allocation from the command line and evaluates it, particularly useful for exhaustive slack allocation;
- `greedy_ra/` – the source for CQSA and derivative Greedy Slack Allocation algorithms;
- `mcs/` – the source for our system modeling framework and lifetime evaluation algorithm;
- `utilities/` – the source for a variety of supporting utilities that have been full integrated, including (a) the Parquet floorplanner [7], (b) the ORION router power estimator [8], and (c) the HotSpot temperature estimator [9].

IV. Using CQSA

There are four steps to using CQSA:

1) creating input files that describe the application and baseline architecture,
2) identifying and programming critical quantities,
3) running CQSA,
4) interpreting the output.

In this section, we will use an example to discuss each of these steps in turn.

A. Example

For our example, we will use a hypothetical implementation of the Multi-window Display algorithm [10], illustrated in Figure 1. The input files for this example can be found in `benchmarks/mwd_old`.

In the task graph (Figure 1a), clouds are execution tasks, and cylinders are storage tasks. Communication between tasks is indicated with arrows. In the architecture (Figure 1b), squares are processors, rounded rectangles are memories, and circles are switches.

We call the architecture in this figure the *baseline architecture*, because it allocates no slack: each resource has been selected to match the requirements of the execution and storage tasks mapped to them. All of the processors are ARM9s, and all of

[3]If under OS X GSL is *not* installed in `/sw/`, or under Linux GSL is *not* installed in `/usr/local/`, you'll have to manually update `Makefile.defs.template` (saving it as `Makefile.defs`) to correctly point to GSL on your workstation. Currently, the package doesn't link properly with GSL on OS X 10.6 using a 64-bit architecture.

the memories are 1MB SRAMs. All execution tasks (clouds) are assumed to be 250 MIPS; task EX is mapped to processor (square) PX. All storage tasks (cylinders) are assumed to be 1MB; task SX is mapped to memory (rounded rectangle) MX.

The critical quantities for each switch are indicated with the shaded boxes around the components connected to each switch in Figure 1b. This architecture has two unique critical quantities: $\langle 750, 0\rangle$ for Switch 3, which will be the basis for Stage 1, and $\langle 500, 1024\rangle$ for Switches 1-2 and 4, which will be the basis for Stage 2.

In this example, execution slack is allocated by replacing ARM9s (250 MIPS) with ARM11s (500 MIPS). Likewise, storage slack is allocated by replacing 1MB SRAMs with 2MB SRAMs.

B. Input Files

CQSA requires as inputs:

- a description of a performance-constrained application, including computation, storage and communication requirements for each software task;
- a fixed communication architecture for a single-chip multiprocessor, including an initial selection of processors, memories, switches and their interconnection; and,
- an initial task-resource mapping, including an assignment of computational tasks to processors, storage tasks to memories, and communication to links and switches.

To facilitate reuse and accommodate internally used tools, these inputs are divided into three files:

- the *task graph* file, which enumerates the tasks in the system, lists the computational or storage requirements of each, as well as the communication that occurs between pairs of tasks;
- the *architecture* file, which enumerates the components in the system, each selected from an internal component library, and assigns tasks from the task graph to each; and,
- the *netlist* file, which specifies how components in the architecture file are initially interconnected.

Together, these files fully specify the initial configuration of the application and architecture. The formatting requirements for each of these files is detailed in the Appendix.

C. Identifying and Programming Critical Quantities

After the application and architecture have been specified, the next step is manually identifying the critical quantities of slack to be explored. In our example, there are two switch critical quantities, $\langle 750, 0\rangle$ for Switch 3, and $\langle 500, 1024\rangle$ for Switches 1-2 and 4.

Critical quantities must be specified by hand in `greedy_sa/src/greedy.cpp`. A large section of the `main` function (starting at line 509) is dedicated to specifying the various critical quantities of the example benchmarks available in `benchmarks/`. Which critical quantities are explored is determined using a set of `#define` statements at the top of `greedy.cpp`. To exercise the critical quantities for our example, `#define APP MWD_OLD` and `#define ARCH 4` should be uncommented, indicating that the set of critical quantities corresponding to the combination of application `MWD_OLD` and its implementation using four switches.

As indicated in Algorithm a:cqsa and discussion above in Section II-B, critical quantities are stored in two lists, one for critical quantities of execution slack alone , and another for critical quantities of execution slack and storage slack. In `greedy.cpp` these lists are called `criticalProcessorList` and `criticalPMList` respectively.

```
vector<int> criticalProcessorsList;
vector<pair<int,int> > criticalPMList;
```

The critical quantities for our example have been specified starting at line 676 of `greedy.cpp`. First, the execution-slack-only critical quantity (covering Switch 3 in Figure 1b) is specified by pushing es onto the list:

```
criticalProcessorsList.push_back(750);
```

Since in this case, $ss = 0$, only es is needed to fully specify the critical quantity. Next, the execution and storage slack critical quantity (cover Switches 1, 2 and 4 in Figure 1b) is specified. A re-usable pair element `pair<int, int> p` has been declared above, for use in constructing the list. `p` is set to $\langle 500, 1024\rangle$ and then pushed onto the list:

```
p.first = 500;
p.second = 1024;
criticalPMList.push_back(p);
```

Our research has shown that considering critical quantities beyond those required to cover network switches can the lifetime-cost optimization results of CQSA, though at the cost of increased computational complexity. For more information, see Chapter 5 and 6 in *Cost-effective Lifetime and Yield Optimization for NoC-based MPSoCs* [6].

D. Running CQSA

Once the desired critical quantities have been specified, use `make` to build `greedy`.

The basic CQSA command-line is structured as follows:

```
./greedy [ parameters ] -t [ task graph ] -c [ architecture ] -n [ netlist ]
```

A variety of parameters are available to modify how CQSA runs. In general, parameters are specified in flag, value pairs, e.g., `-u 0`. The most basic set of parameters are reviewed in table I.

TABLE I: Commonly Used CQSA Parameters

Flag	Value	Default	Description
`i`	0 or 1	0	1: enable use of initial component temperatures in lifetime estimation
`u`	0 or 1	0	1: enable updating of component temperature when components fail, requires `-i 1`
`y`	0 or 1	0	1: estimate system yield rather than system lifetime
`a`	0 or 1	0	1: floorplan to minimize area; 0: floorplan to minimize weighted sum of area and wire length
`f`	int > 0	100	the number of floorplanning iterations used; more iterations results in better floorplans
`s`	int > 0	1000	the number of lifetime estimation iterations used; more iterations results in more accurate lifetime estimates
`d`	db file	null	import a database of results to accelerate execution

For our example, we use the following command line:

```
./greedy \
        -u 0 -i 0 -a 1 -f 100 -s 10000 \
        -c ../benchmarks/mwd_old/config/4-s.cfg \
        -n ../benchmarks/mwd_old/fp/4-s.nets \
        -t ../benchmarks/mwd_old/$mwd_old.tg \
        -d ../benchmarks/mwd_old/4-s.db
```

The `-d` flag uses a database of previously computed results (in this case, exhaustively determined) to accelerate the execution of CQSA. In this case, when a new design is encountered, the database is queried for lifetime and cost data for that particular design. If it exists, the pre-computed values are used. Otherwise, lifetime and cost are evaluated.

E. Output

The resulting output of CQSA can be found in `greedy_sa/results/example/mwd_4-s.out`. After some basic header information, the results detail each design configuration that is evaluated, including the maximum capacity of each component in the system, system area, system wire length and average system lifetime.

The results are reported in the same order as designs are encountered in CQSA. Naturally, Stage 0 is the first to be evaluated, followed by Stage 1 and Stage 2. Within each stage, the designs are broken into groups based on the amount of slack in the system. The first design reported is the baseline design:

```
blnd hs hvs in jug1 jug2 nr se vs mem1 mem2 mem3 area wl mttf
250 250 250 250 250 250 250 250 250 1024 1024 1024 54.37 89.7755 9.28105
```

The header line indicates the names of the components listed below, and labels the results fields. The rest of the results in the file are divided up by the amount of slack considered. For example, when considering 250 MIPS in stage 0, the output is:

```
###   250 MIPS
500 250 250 250 250 250 250 250 250 1024 1024 1024 55.7025 71.6575 10.5249
250 500 250 250 250 250 250 250 250 1024 1024 1024 55.7025 47.7255 10.5207
250 250 500 250 250 250 250 250 250 1024 1024 1024 55.7025 49.9575 10.3707
250 250 250 500 250 250 250 250 250 1024 1024 1024 55.7025 52.0175 9.75662
250 250 250 250 500 250 250 250 250 1024 1024 1024 55.7025 52.3355 10.3679
250 250 250 250 250 500 250 250 250 1024 1024 1024 55.7025 62.4255 10.3716
250 250 250 250 250 250 500 250 250 1024 1024 1024 55.7025 43.8855 10.3615
250 250 250 250 250 250 250 500 250 1024 1024 1024 55.7025 53.7255 10.5206
250 250 250 250 250 250 250 250 500 1024 1024 1024 55.7025 54.1855 10.3694
*** 500 250 250 250 250 250 250 250 250 1024 1024 1024 55.7025 71.6575 10.5249
```

Every possible allocation of 250 MIPS is explored and reported (there are nine). The final line, marked with `***` repeats the selected configuration, the design with the best MTTF.

Figure 2 plots the lifetime (1/MTTF, 1/years) and cost (mm^2) of the baseline design (A) and the designs selected in Stage 0 (red squares), Stage 1 (green triangles), and Stage 2 (blue x's). The gray line marks the Pareto-optimal front.

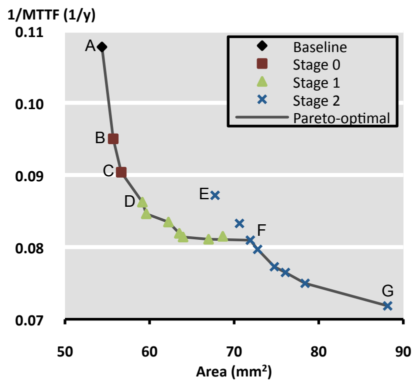


Fig. 2: The selected design points from a CQSA run on the baseline architecture in Figure 1b. Stage 0 greedily allocates slack to the baseline (A), ultimately resulting in C. Stage 1 continues to allocate execution slack, but informs its initial slack allocation (D) by considering the critical quantity $\langle 750, 0\rangle$. Stage 2 begins with another critical quantity, $\langle 500, 1024\rangle$ (E), but doesn't find a Pareto-optimal design until 500 MIPS more have been allocated (F).

References

[1] J. Srinivasan, *et al.*, "The impact of technology scaling on lifetime reliability," in *DSN'04*, June 2004.
[2] J. C. Smolens *et. al*, "Detecting emerging wearout faults," in *SELSE-3*, April 2007.
[3] C. Zhu, *et al.*, "Reliable multiprocessor system-on-chip synthesis," in *CODES+ISSS'07*, October 2007.
[4] B. H. Meyer, A. S. Hartman, and D. E. Thomas, "Cost-effective slack allocation for lifetime improvement in NoC-based MPSoCs," in *the Proceedings of the 2010 Conference on Design, Automation, and Test in Europe, DATE'10*, March 2010.
[5] B. H. Meyer, A. S. Hartman, and D. E. Thomas, "Slack allocation for yield improvement in NoC-based MPSoCs," in *the Proceedings of the 11th annual International Symposium on Quality Electronic Design, ISQED'10*, March 2010.
[6] B. H. Meyer, *Cost-effective Lifetime and Yield Optimization for NoC-based MPSoCs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2009.
[7] S. N. Adya and I. L. Markov, "Fixed-outline floorplanning: Enabling hierarchical design," *IEEE Transactions on VLSI Systems*, vol. 11, December 2003. http://vlsicad.eecs.umich.edu/BK/parquet/.
[8] H. Wang, *et al.*, "A power model for routers: Modeling Alpha 21364 and InfiniBand routers," *IEEE MICRO*, vol. 23, Jan./Feb. 2003.
[9] K. Skadron *et al.*, "Temperature-aware microarchitecture," in *ISCA'03*, June 2003.
[10] E. G. T. Jaspers and P. H. N. de With, "Chip-set for video display of multimedia information," *IEEE Trans. on Consumer Electronics*, vol. 45, August 1999.
[11] http://www.arm.com/products/CPUs/families.html.
[12] "CACTI 5.3." http://www.hpl.hp.com/research/cacti/.

Appendix

This appendix specifies the file formatting requirements for input files and documents CQSA's internal component library.

A. Task Graph File Format

The task graph file specifies all tasks in the application, their sizes, and how they communicate. The initial assignment of tasks to resources is specified in the architecture file.

There are two sections to the task graph file, the computation section, which specifies the name and size of tasks in the application, and the communication section, which specifies the communication between tasks in the application.

The computation section is declared with `define computation` and closed with `end`. Each entry in the computation section defines an internal task name (a string) followed by its size (an integer) using the following syntax:

```
< task name > < task size >
```

An example computation section is as follows:

```
define computation

# computational requirement in MIPS
in 125
nr 500

# storage requirement in kB
mem1 1024
end
```

The above code defines three tasks, `in`, `nr` and `mem1`, and specifies their sizes as 125, 500, and 1024. Comments (indicated with #) indicate that in and nr are intended to be execution tasks, while mem1 is intended to be a storage task. The task graph file doesn't explicitly differentiate between execution and storage tasks; the initial assignment of a task to a resource in the *architecture* file determines the type of the task (i.e., it is a computation task if it is assigned to a processor, a storage task if it is assigned to a memory).

The communication section is declared with `define commiunication` and closed with `end`. Each entry in the communication section defines the communication requirement (in KB/s) between two tasks already defined in a computation section, using the following syntax:

```
< src task > < dst task > < KB/s >
```

Each communication requirement is assumed to be unidirectional. An example communication section is as follows:

```
define communication

# communication requirement in kB/s
in nr 65536
nr mem1 65536
mem1 nr 65536

end
```

This defines three communication channels, from `in` to `nr`, another from `nr` to `mem1`, and another from `mem1` back to `nr`.

Additional examples can be found in the distribution, in `zzz.tg` files within `benchmarks/zzz/`, where `zzz` is the name of the benchmark.

B. Architecture File Format

The architecture file specifies all of the components (processors, memories and switches) in the system, how tasks (defined in the task graph file) are initially assigned to them, and any failure dependencies that may exist. Components are selected from the internal component library. The netlist file specifies how resources are connected.

There are two sections in the architecture file, the components section, which specifies the name and task assignment of each resource in the architecture, and the preclusions section, which specifies any failure dependencies in the architecture.

The component section is declared with `define components` and closed with `end`. Entries in this section specify the internal name of the component, its component type, and the number of tasks assigned to it, and a list of assigned tasks, using the following syntax:

```
< name > < type > < # of tasks > < task name [ task name [ ... ]]>
```

No tasks are ever mapped to switches; the number of tasks assigned to a switch is therefore specified as 0. An example components section is as follows:

```
define components

in M3 1 in
nr M3 1 nr

mem1 MEM1MB 1 mem1

s_nr SW3X3 0

end
```

The above code defines four resources, `in`, `nr`, `mem1`, and `s_nr`. `in` and `nr` are declared to be of type `M3`, a processor, while `mem1` is declared to be of type `MEM1MB`, a 1MB SRAM. Each of these resources are assigned a single task that bears the same name (not a requirement, but also not illegal). `s_nr` is declared to be of type `SW3X3`, a switch with a 3x3 crossbar.

The preclusions section is declared with `define preclusions` and closed with `end`. Each entry in the preclusions section indicates, when a particular component failures, which other components become inaccessible. An example would be that, when a switch fails, any resources connected only to that switch becomes inaccessible. List such resources in the preclusion section marks the inaccessible resources as failed so that their tasks can be re-mapped and their traffic re-routed.

The syntax for an entry is as follows:

```
< failed component > < inaccessible component > [ < inaccessible component > ... ]
```

Additional examples can be found in the distribution, in `.cfg` files within `benchmarks/zzz/config/`, where `zzz` is the name of the benchmark. There may be many architecture files associated with a single task graph. In general, the name for the architecture file will be related to that of the corresponding netlist (e.g., `2-s.cfg` and `2-s.nets` are used together).

C. Netlist File Format

The netlist file specifies all of the links between resources in the system. In general, any resource may be connected to any other resource, though traffic may only be routed through switches. The format for the netlist differs from that of the task graph and architecture files, as it has been tailored for use with the Parquet floorplanner [7]. The file begins with a simple header:

```
UCLA nets 1.0

NumNets : < number of nets >
NumPins : < number of pins >
```

This distribution only supports nets with two pins; as a result, $\texttt{NumPins} = 2 \cdot \texttt{NumNets}$.

After the header, the rest of the file specifies each link in the communication architecture. Links may be specified as uni-directional or bi-direction, using the following syntax:

```
NetDegree : 2 [ net name ]
        < src resource name > < U, B >
        < dst resource name > < U, B >
```

`U` indicates a unidirectional link, while `B` indicates a bi-directional link. Each resource must indicate the same directionality: both must either be `U` or `B`. In the case of bi-directional links, whether a resource is the source or destination is irrelevant.

Examples netlists can be found in the distribution, in `.nets` files within `benchmarks/zzz/fp/`, where `zzz` is the name of the benchmark. There may be many architecture files associated with a single task graph. In general, the name for the netlist file will be related to that of the corresponding architecture (e.g., `2-s.cfg` and `2-s.nets` are used together).

D. Component Library

Each architecture is implemented using processors, memories and switches from a small, internal component library. The information required to describe a component in the internal component library is defined in `mcs/include/ComponentLibrary.h`, while the different available components are specified in `mcs/src/ComponentLibrary.cpp`. Though the creation of new components within this framework is beyond the scope of this report, it should not be beyond the ability of an average C++ programmer.

We use three types of processors, the ARM Cortex M3 (M3), ARM946E-S (ARM9), and ARM1156T2(F)-S (ARM11). The relevant properties of these processors were determined using datasheet values and are summarized in Table II [11]. Execution slack is allocated in the systems we consider by replacing M3s (125 MIPS) with ARM9s (250 MIPS) or ARM11s (500 MIPS), or by replacing ARM9s with ARM11s. No execution slack can be allocated to a node that is already an ARM11.

TABLE II: Component Library: Processors

Name	Part	Area (mm^2)	Cap (MIPS)	Power (mW/MIPS)
M3	ARM Cortex M3	0.61	125	0.10
ARM9	ARM946E-S	1.00	250	0.17
ARM11	ARM1156T2(F)-S	2.40	500	0.51

We use nine sizes of SRAMs, from 64 KB to 2 MB. The relevant properties were determined using the low-standby-power model in CACTI 5.3 and are summarized in Table III [12]. Storage slack is allocated in the systems we consider by replacing a memory with a higher capacity memory (e.g., replacing a 192 KB SRAM with a 384 KB SRAM). No storage slack can be allocated to a node that is already a 2 MB SRAM. In practice, it is sometimes appropriate to limit memory nodes to less than 2 MB in order to manage the size of the design space.

TABLE III: Component Library: Memories

Name	Part	Area (mm^2)	Cap (KB)	Read (nJ)	Write (nJ)	Leakage (mW)
MEM64KB	64 KB SRAM	1.02	64	0.143	0.113	0.0092
MEM96KB	96 KB SRAM	1.51	96	0.121	0.098	0.0137
MEM128KB	128 KB SRAM	1.90	128	0.143	0.113	0.0183
MEM196KB	192 KB SRAM	2.89	192	0.216	0.146	0.0263
MEM256KB	256 KB SRAM	3.68	256	0.245	0.158	0.0350
MEM384KB	384 KB SRAM	5.20	384	0.316	0.188	0.0523
MEM512KB	512 KB SRAM	6.72	512	0.386	0.216	0.0698
MEM1MB	1 MB SRAM	13.19	1024	0.462	0.291	0.1400
MEM2MB	2 MB SRAM	25.14	2048	1.210	0.418	0.2730

Our component library also contains several network-on-chip switches. Because of the area and power consumption of large switches, we limit switch crossbars to no larger than 5x5. As a result, our component library contains only three types of switches, 3x3 switches, 4x4 switches and 5x5 switches. Each switch is equivalent to the Alpha 21364's on-chip router. The area for each switch is detailed in Table IV; the power consumption for each system switch is determined dynamically based on its load using ORION 1.0 [8].

TABLE IV: Component Library: Switches

Name	Part	Area (mm^2)
SW3X3	3x3 Switch	0.35
SW4X4	4x4 Switch	0.47
SW5X5	5x5 Switch	0.59