Selected Papers from the Proceedings of the Fourth Student Symposium on Computer Systems (SOCS-4)

Theodore Wong (*Editor*)

October 6, 2001

CMU-CS-01-164

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

SOCS provides a forum for Carnegie Mellon computer systems students to present recent work and new ideas to their peers. We present selected papers from the Proceedings of SOCS-4, covering research in compilers, scheduling, networking, and security. We also present selected works-in-progress.

Keywords: compilers, distributed systems, networking, security, operating systems

Executive Committee

General Chair	Theodore Wong
Program Chair	Mihai Budiu

Program Committee

Pedro Vaz Artigas, CS
Angela Demke Brown, CS
Yang-hua Chu, CS
Jun Gao, CS
Andrew Klosterman, ECE
Ben Levine, ECE
Julio Lopez, ECE
William Nace, ECE
David Petrou, ECE
Sanjay Rao, CS
Steve Schlosser, ECE
Craig Soules, ECE
Mengzhi Wang, CS

External Reviewers

Chris Colohan, CS
Kevin Watkins, CS

Special Thanks

Greg Ganger
Karen Lindenfelser

Message from the General Chair

SOCS continues to build on the success of the previous years. Now in its fourth year, SOCS has again attracted considerable interest from students wishing to showcase their ongoing research. The community looks forward to seeing their work appear in external conferences.

I would like to take this opportunity to acknowledge the many people who helped to make SOCS possible. I would like to thank Mihai Budiu, the program committee, and the external reviewers for their efforts in reviewing the submitted papers and putting together an interesting program. I also thank Jason Flinn and Jiri Schindler (the SOCS-3 general and program chairs respectively) for passing on their experiences. And of course, I would like to thank Greg Ganger for his generous financial sponsorship of SOCS.

Theodore Wong
General Chair

Message from the Program Chair

With great pleasure I welcome you to the fourth annual edition of CMU's Symposium on Computer Systems, SOCS. SOCS is a conference run by students: students submit the papers, they compose the program committee, they are the reviewers, they are in charge of publicity, printing and all arrangements. SOCS is thus both an occasion to prepare for "real-world" conferences from all points of view, and a forum to present one's research to the CMU community.

This year's SOCS had 12 paper submissions, and the committee had to work very hard to make a selection. In the end, we have 8 full papers, grouped in three main categories: computer architectures, compilers and scheduling and networking and security.

I want to express my gratitude to all the program committee members, who have very promptly responded with the assigned tasks, making running this conference an easy endeavor. I also thank warmly all the authors who have submitted: the conference is as good as the papers we receive, and they have been outstanding.

The committee has agreed to reward the best of these papers with our "Best Paper Award". This year the award goes to Chris Gniady, for his paper *Speculative Sequential Consistency with Little Custom Storage*; his advisor is Babak Falsafi.

I am looking forward to next year's SOCS.

Mihai Budiu
Program Chair

Table of Contents

Application-Specific Hardware: Computing Without CPUs

Mihai Budiu
mihaib@cs.cmu.edu

Abstract

In this paper we propose a new architecture for general-purpose computing which combines a reconfigurable-hardware substrate and compiler technology to generate Application-Specific Hardware (ASH). The novelty of this architecture is that resources are not shared: each different static program instruction can have its own dedicated hardware implementation. ASH enables the synthesis of circuits with only local computation structures, which promise to be fast, inexpensive and use very little power. This paper also presents a scalable compiler framework for ASH, which generates hardware from programs written in C and some evaluations of the resources necessary for implementing realistic programs.

1 Introduction

For five decades the relentless pace of technology, expressed as Moore's law, has supplied computer architects with ample materials in the quest for high performance. The abundance of resources has translated into increased complexity[1]. This complexity has already become unmanageable in several respects:

- The verification and testing cost escalates dramatically with each new hardware generation.
- Manufacturing costs (both plant costs and non-recurring engineering costs) have skyrocketed.
- Defect density control becomes very expensive as the feature size shrinks; in the near future we will be unable to manufacture large defect-free integrated circuits.
- The dissipated power density (watts/mm^2) of state-of-the-art microprocessors has already reached values that make air-cooling infeasible [4].
- The clock frequency has increased to a value where global signals across the entire chip are infeasible (the propagation delay exceeds the clock cycle [1]).
- The number of exceptions which require manual interventions generated by the CAD tools grows quickly with design complexity [16].
- Today's processors use extremely complicated hardware structures to enable the exploitation of the instruction-level parallelism (ILP) in large windows; however, the sustained performance is rather low [15].

In Section 2 we propose an alternative approach to implement general-purpose computation, which consists of synthesizing — at compile time — application-specific hardware, on a reconfigurable-hardware substrate. We argue that such hardware can be more efficient than a general-purpose CPU, and can solve or alleviate all of the above problems. We call this model **ASH**, from Application-Specific Hardware.

We propose a way to synthesize directly custom, application-specific dataflow machines in hardware. The ASH machines have low overhead, as they implement the whole application in reconfigurable hardware, and avoid time-multiplexing hardware resources.

The main component of the ASH framework is **CASH**, a Compiler for ASH, presented in Section 3. CASH spans both the realm of traditional compilation and hardware synthesis.

In Section 4 we evaluate the hardware resources needed to implement realistic programs within the ASH model of computation. Section 6 describes some implications of the ASH architecture on computer system design.

2 Application-Specific Hardware

In this section we give an overview of the ASH model of computation. The core of ASH is a reconfigurable fabric; compilation subsumes the role of traditional software compilation and hardware synthesis, translating high-level language programs into hardware configurations.

Reconfigurable hardware devices are hardware devices whose functionality can be changed dynamically (see [10] for a survey). The most common type of device is a Field-Programmable Gate Array and features a set of universal logic gates connected by a switched interconnection network. The logic gates are implemented as look-up tables from small memories; by changing the contents of each memory we change the function computed by each gate. Configuration bits also control the switches on the interconnection network; by choosing which switches are

[1] In this paper we will be mostly concerned with the complexity of microprocessors.

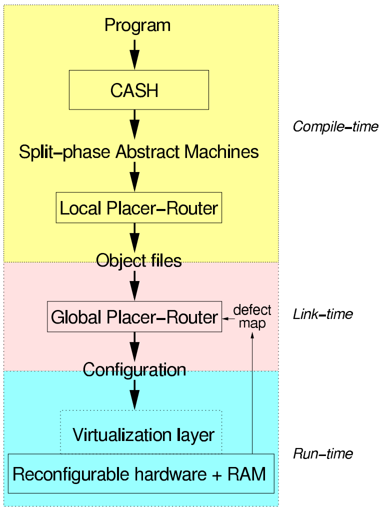


Figure 1: *The ASH tool-flow.*

open and which are closed we effectively connect the logic gates to each other. Reconfigurable hardware thus features the flexibility of general-purpose programmable systems and computation speeds comparable to raw hardware.

Figure 1 summarizes our framework. Programs written in general-purpose high-level languages are the input to the CASH compiler. After applying traditional program-optimization techniques, CASH decomposes the program into small fragments, called Split-phase Abstract Machines, or SAMs.

Each SAM is optimized, synthesized, placed, and routed independently. The placed SAMs that compose the complete program are fed to a global placer and router[2] which decides how to lay-out the machines and how to connect them using an interconnection network. The resulting "executable" is a configuration for the reconfigurable hardware. At run time the configuration is loaded on the reconfigurable-hardware substrate and executed. If the configuration is too large, a run-time hardware-virtualization method may be used.

In this paper we only present the CASH component from Figure 1.

2.1 Split-phase Abstract Machines

The Split-phase Abstract Machine (SAM) is the main abstraction of our intermediate program representation. The compiled program is partitioned into a collection of SAMs, which communicate asynchronously with each other. Each SAM contains computation and possibly a small local memory. The computation implemented in a SAM has predictable latency[3]; moreover, the SAM local memory has predictable access times.

SAMs are inspired by the Threaded Abstract Machine model [11]; like TAMs, whenever a SAM needs to execute an operation that has unpredictable latency it uses the inter-SAM communication network: remote memory accesses, and control-flow transfers between SAMs are transformed into messages routed dynamically on the network. SAMs roughly approximate the procedures in a high-level programming language (however, in our implementation a procedure can be decomposed into several SAMs).

During the program execution, at each instant a SAM can be in one of three states:

- **Inactive SAMs** are not being executed and do not have any live state. These SAMs do not need to consume any power and, if hardware virtualization is available, can be swapped out of the reconfigurable hardware.
- **One active SAM** is actively switching and consuming power and should be entirely swapped in; it is analogous to the procedure on the top of the stack (currently being executed) in a traditional model of computation[4].
- **Passive SAMs** are mostly quiescent: they store live values, but are blocked waiting for the completion of a "callee" SAM. They dissipate only static power most of the time[5] and correspond roughly to the procedures in the current call chain, which have been started in execution, but have not been completed.

2.2 An example

Figure 2 shows a simple C program and the equivalent translation into three SAMs. This figure has been automatically generated by an early version of our prototype CASH compiler using the VCG graph layout tool [18] as a back-end. This figure illustrates just one possible implementation, and a rather suboptimal one.

The compiler creates three SAMs from this program:

SAM 1 implements the initialization of the variables `i` and `j` with 0. It receives as input the "program counter" (PC), which indicates the caller SAM. The shaded empty oval receives a control token which enables the current SAM to start execution. The lightly shaded rectangles with a sharp sign are output registers, containing data that is passed to SAM 2.

[2]The global placer can use a defect map of the target chip to provide fault-tolerance, by avoiding the defective regions.

[3]SAMs can also invoke remote operations, which have unpredictable latencies.

[4]Currently we only consider programs which have a single thread of execution; a parallel model of execution might have several active SAMs at one moment.

[5]There may be some concurrent activity between the passive SAMs and the active one, because of "instructions" that can be executed in parallel with the "call".

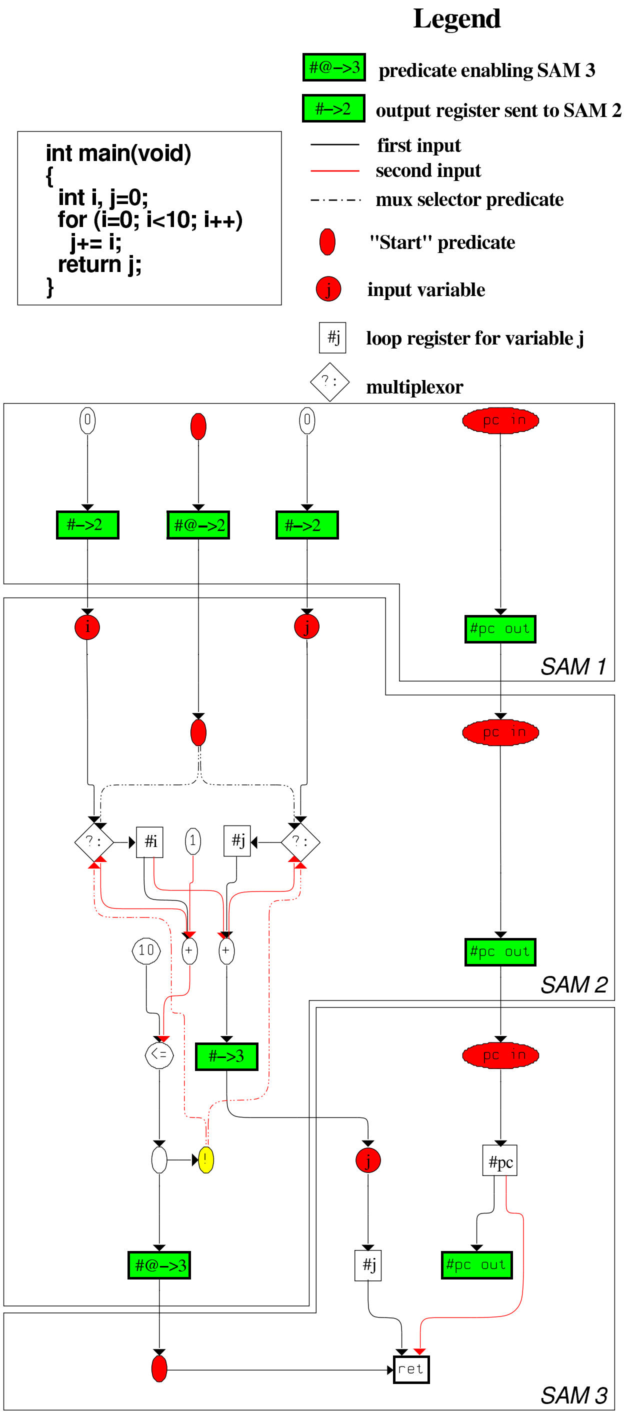


Figure 2: *A simple C program and its equivalent SAM implementation. The tokens are not explicitly represented in this figure: conceptually each wire has an associated token, indicating when the signal on the wire is valid.*

SAM 2 implements most of the computation in the procedure. It contains two additions, one for `i` and one for `j`, a comparison of `i` with 10, and two multiplexors (represented by the `?:` diamonds) that select the values for `i` and `j` based on the flow of control: either the initial value or the result from the increment operation. The multiplexors have two data inputs and two control inputs (dotted lines) each; the shades correspond: when the dark dotted line is asserted, the dark input is selected. The boxes marked with sharp signs # are registers holding the state, represented by the values of `i` and `j`.

SAM 2 is executed as long as the loop condition is true (i.e., `i < 10`). When the loop condition becomes false, control is transfered to the SAM 3.

SAM 3 executes just the `return` instruction. It receives from SAM 2 the value of `j` and the PC and uses them as arguments to the return "instruction". Because the return instruction has a side-effect, it has a third input, a predicate, which indicates when the instruction is safe to execute. Because this return is executed unconditionally, the predicate is fed directly from the enabling token. This return instruction uses the PC value to return the control to the SAM that had originally invoked SAM 1.

2.3 Benefits of the ASH Model

The ASH model has better scalability properties than traditional CPU architectures. For instance:

- The verification and testing of a homogeneous reconfigurable fabric is much simpler. The program is translated directly into hardware, so there's no interpretation layer (i.e., the CPU) which can contain bugs. Moreover, we believe that by building CASH as a certifying compiler [23][6], we can completely eliminate one complex layer needing verification and testing (the processor).
- The manufacturing of reconfigurable circuits reuses the same masks for all circuits, reducing cost.
- As shown by research in the Teramac project [13], reconfigurable hardware architectures can tolerate manufacturing defects through software methods.
- Only the active SAM is switching at any point, requiring very little power.
- The SAM implementation uses only local signals. All inter-SAM communication is made using a switched, pipelined interconnection network. There is no need for global electrical signals.
- CAD tools for reconfigurable hardware can be much simpler than general VLSI tools.
- Dynamic methods of extracting ILP from programs (as implemented in today's out-of-order processors)

[6] A certifying compiler generates not only an executable but also a formal proof that the executable is equivalent with the input program.

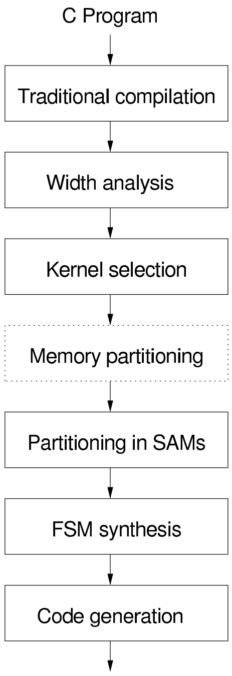


Figure 3: *The CASH compiler passes. The dotted line indicates a component which is not implemented.*

are hindered by limited issue windows: they cannot exploit parallelism outside of a relatively small window of instructions. Our compiler analyzes large program fragments and can uncover substantially more parallelism. We quantify the ILP we discover in Section 4.

The main disadvantage of the ASH paradigm is the requirement for substantial hardware resources. However, this can be alleviated through use of virtualization, or by hardware-software partitioning between a CPU and an ASH fabric. We will quantify the resources necessary in Section 4.

The interaction of the ASH model of computation with the operating system and with multi-tasking is a subject of future research.

3 CASH

In this section we describe CASH, our current implementation of the ASH Compiler. Our compiler infrastructure is built around the SUIF 1.3 research compiler [35]. For the moment, we do not use any of the parallelizing components of SUIF. Figure 3 illustrates the main passes of our compiler. Here are brief descriptions of each of them:

Traditional compilation: most standard compiler front-end optimizations (e.g., dead-code elimination, copy propagation, common subexpression elimination, unreachable code removal) are beneficial in the context of the ASH framework. We also use aggressive procedure inlining.

Width analysis and hardware cost estimation: our reconfigurable hardware target can implement arbitrary-width arithmetic efficiently. We use the BitValue [6] analysis algorithm, which can discover narrow-width scalar operations in C programs. The results presented in this paper do not make use of BitValue, but we plan to incorporate them in future work.

Kernels selection: when we target a system comprised of a CPU and a reconfigurable system, this pass selects program portions that are most likely to provide benefits when executed on the reconfigurable fabric. In the rest of this paper however, we assume that we compile the whole program in reconfigurable hardware, so we do not use any kernel selection algorithm.

Memory partitioning: a variety of techniques can be employed to discover for each piece of code the memory regions that it will access. Once this kind of information is available, various techniques can be used to co-locate the memory and the code accessing it. This part of the compiler is currently not implemented: all memory accesses are made to an external monolithic memory, like in CPU-based systems.

SAM selection: the compiled program is decomposed into split-phase abstract machines. In order to extract a large amount of ILP we implement implement each SAM from one hyperblock [21]. Hyperblocks have been introduced in the context of predicated-code machines, and comprise multiple program basic blocks. We discuss this phase in detail in Section 3.1.

FSM synthesis: from each SAM we generate a finite-state machine (FSM). The FSM has a combinational portion, which computes the next state and a feedback portion, which implements the looping. The FSM state consists of the loop-carried variables. We discuss this compilation phase in detail in Section 3.2.

Code generation: the only back-ends we have implemented so far are a graph-drawing back-end (which was used to generate Figure 2) and a generator which outputs C programs. These programs simulate the execution of the SAMs and compute timing information. In the future, we plan to adapt the back-end to generate HDL descriptions of the SAMs.

3.1 SAM Implementation

In this section we present details about our implementation of the Split-phase Abstract Machines.

The main abstraction we use at the program level is the hyperblock [21]. A hyperblock is a part of the program control-flow graph (CFG) that has a single entry point but possibly multiple exits. The hyperblock may contain loops, but all the loops have to share the same loop entry point, which is also the hyperblock entry point. In our current implementation, each hyperblock becomes a SAM.

Hyperblocks have already been used for generating reconfigurable-hardware implementations in [8, 27]. Our method of hyperblock selection is completely general and deals with unstructured flow of control. We cover each procedure with disjoint hyperblocks, using a linear-time algorithm, as follows:

- A depth-first traversal of the CFG is used to label the back-edges;
- End-points of the back-edges are marked as hyperblock entry points;
- A reverse depth-first traversal is used to assign basic blocks to hyperblocks, as follows:
  - a basic block is in the same hyperblock as all its predecessors;
  - unless two predecessors are from different hyperblocks, in which case the block is itself a hyperblock entry point.

There are several knobs that we can turn to tune the hyperblock selection. We can optimize the resulting circuit for area, speed, or power. One degree of freedom is the traversal order, which defines the back-edges and thus the entry points. A second degree of freedom we have is to add extra entry points, fragmenting large hyperblocks into several small ones. A third degree of freedom is the possibility of duplicating the body of the basic blocks that have multiple hyperblock predecessors and thus creating fewer large hyperblocks (this last technique is used in [21]). We have not explored any of these trade-offs. Our hyperblock selection scheme generalizes all the other proposed schemes[7].

Some hyperblocks will contain loops. Because these loops have exactly one entry point, which coincides with the hyperblock entry point, they are well-structured and the loop-induction variables are well-defined. We can thus synthesize each hyperblock into a finite-state machine.

3.2 FSM Synthesis

We implement each finite-state machine as a dataflow machine [33]. In dataflow machines the computation is expressed as a graph of operations which are triggered by the availability of data. Each data item is encapsulated within a "token", which indicates the functional units that are supposed to process it; when all the inputs of a functional unit are available, the unit consumes the tokens and generates a new one.

In our implementation, the "tokens" are no longer explicitly represented: they become two 1-bit signals connecting the functional units (one bit is used to signal data availability, the other to confirm data consumption). There is no token store, token-matching logic or register file. The main overhead of the interpreted dataflow machines is thus completely eliminated. Static scheduling can eliminate most of the token synchronization.

The computation of the combinational portion of the FSM is implemented speculatively in the style of predicated static-single assignment [9] and predicated speculative execution [2]. To illustrate the implementation, we use the example in Figure 4, a code snippet from the `g721` Mediabench [17] program.

3.3 Path Predicates

Each basic block in a hyperblock has an associated *path predicate*, as described in [9]; the path predicate associated to block B is true if and only if block B is executed during the current loop iteration. The predicates corresponding to blocks are recursively defined:

$$\begin{array}{l} P(entry) = True \\ P(s) = \vee_{p \in Pred(s)} (P(p) \wedge B(p,s)) \end{array} \quad (1)$$

where $B(p,s)$ is true if block p branches to s. This should be read as: "Block s is executed if and only if one of its predecessors p is executed and p branches to s." For the example in Figure 4:

$$\begin{array}{rcl} P(a) &=& True \\ B(a,b) &=& (\texttt{fa1} < -8191) \\ B(a,c) &=& \neg B(a,b) \\ P(b) &=& P(a) \wedge B(a,b) \\ P(c) &=& P(a) \wedge \neg B(a,b) \\ B(c,d) &=& (\texttt{fa1} > 8191) \\ P(d) &=& P(c) \wedge B(c,d) \\ P(e) &=& P(c) \wedge \neg B(c,d) \\ B(b,f) &=& True \\ P(f) &=& (P(b) \wedge B(b,f)) \vee (P(d) \wedge B(d,f)) \\ && \vee (P(e) \wedge B(e,f)) \end{array}$$

We next use a method equivalent to the *instruction promotion* technique described in [21], which removes predicates from some instructions or replaces them with weaker predicates, enabling their speculative execution. Interestingly enough, if the hyperblock code is in static-single assignment form, we can prove that every instruc-

[7] Other schemes make use of profiling information, which we could easily accommodate.

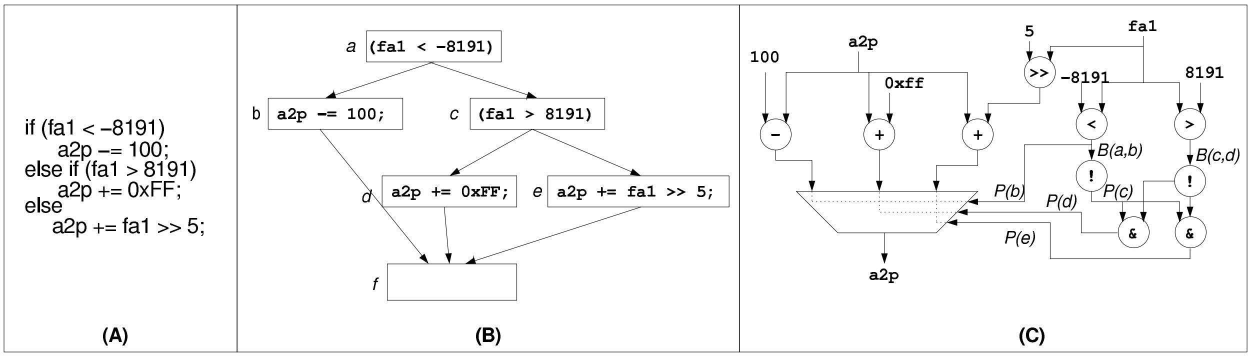


Figure 4: *(A) A code fragment (B) Its control-flow graph (C) Its speculative implementation in hardware.*

tion with no side-effects can be safely and completely promoted to be executed unconditionally.

We will sketch a proof of this fact here. We can distinguish four types of instructions: (a) instructions with side-effects (i.e. memory accesses and procedure calls), (b) predicate computations, (c) multiplexors and (d) all other instructions.

The instructions of type (a) cannot be executed speculatively. We will argue below that instructions of type (b) (i.e. the predicates) can be speculatively computed. It will follow that the multiplexors (c) will always make the correct choices, because they select based on the predicates. It will follow that instructions of type (d) can always be safely executed speculatively, as they have no side-effects and their inputs are always correct.

We now prove that all instructions of type (b), which compute predicate values, can be speculatively executed. The key observation is that, although the predicates are themselves speculatively computed, their value is always correct. We will illustrate the proof using a simple example.

Consider the CFG fragment in Figure 5. According to formula 1, $P(c) = (P(a) \wedge B(a, c)) \vee (P(b) \wedge B(b, c))$. Let us assume that during an execution of the program, basic block a is executed and it branches to c. This means that block b is not executed; thus, the branch condition $B(b, c)$ may have an incorrect value (for instance, because block b changes some values which influence the branch computation). However, by using induction on the depth of the block, we can assume that $P(b)$ has the correct value, False. Thus, the value of $B(b, c)$ is irrelevant for the computation of $P(c)$.

The predicate computation is implemented in hardware, using the same dataflow style. The path predicates are used to guard the execution of instructions with side-effects (memory writes, memory reads that can trigger exceptions, procedure calls and returns). Predicates also control the looping of the FSM; on exit from the current SAM, the predicates also indicate which of the successor

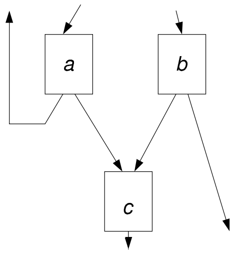


Figure 5: *Fragment of a control-flow graph.*

SAMs should be executed.

This implementation of the program is essentially a static single assignment representation (SSA) [12] of the predicated program. The ϕ functions of the SSA form are essentially multiplexors, selecting among the many definitions of a value that reach a join point in the control-flow graph. Unlike other proposed SSA representations for predicated code [9], we explicitly build the circuitry to compute the ϕ functions, which become multiplexors in hardware (see Figure 4(C)). The multiplexors are controlled by the path predicates. We found that this explicit representation of the complete program is extremely handy, enabling a lot of classical compiler optimizations (dead-code elimination, common-subexpression elimination, constant folding, etc.) to be carried practically in linear time and using very simple and clean algorithms.

The predicates are implemented using formula 1, which implies that their cost is small: each new predicate requires just the logical disjunction of a set of values which were computed previously. Each edge in the CFG contributes one term to the predicate computation, so the implementation of all the predicates together uses resources linear in the size of the hyperblock.

3.4 Eager multiplexors

One problem of the predicated-execution architectures is that the execution time on the speculated control-flow

```
for each basic block (b) in topological order
    L = predecessors(b)
    for each variable (v) live on entry in b
        create a multiplexor M with |L| inputs
        I = 0
        for each block p ∈ L
            V = definer of v at exit of p
            M.input(I) = V
            M.selector(I) = P(p) ∧ B(p, b)
            I = I + 1
        endfor
    endfor
endfor
```

Figure 6: *Algorithm for insertion of multiplexors in a program.*

paths may be unbalanced [3]. For instance, assume that the subtraction takes much longer than the addition; then the leftmost path in Figure 4(C) is the critical path, which dominates the computation time irrespective of which operation should be executed. We have a very simple solution to this problem, which consists of using *eager, fully decoded multiplexors*.

Our multiplexor implementation uses fully-decoded multiplexors, which have as many selector bits as there are inputs. Each selector bit selects one of the inputs, as shown by the dotted lines in Figure 4(C). These multiplexors do not need the complicated encoding/decoding logic for the selection signals and can be very cheaply implemented in hardware, as a wired-or. The eager multiplexor can generate its output as soon as one selector predicate is "True" and the corresponding selected data item is valid[8]

3.5 Multiplexor placement

As we already noted, the placement of multiplexors corresponds to the placement of ϕ functions in SSA form. However, our problem is simpler, because all the back edges in a hyperblock go to the entry point and the rest of the hyperblock can be treated as a feed-forward program fragment. Our current algorithm is presented in Figure 6[9].

We next run a multiplexor simplification pass, which repeatedly uses the following two rules:

[8]Using eager multiplexors might not be enough to guarantee minimal-time execution within a loop. If the speculated paths can be proven statically to be unbalanced we can just avoid executing the long one speculatively or we can use a "reset" signal to abort the computation when it is known that its result is not needed.

[9]For the placement of the multiplexors we plan to implement the algorithm described in [32] which has a much lower complexity. The generated circuit will be the same.

```
procedure merge(m, n)
/* Merge muxes m, n, where m is the k-th input of n */
    remove n's k-th input
    foreach input i of m
        add i as a new j-th input to n
        sel(n, j) = sel(n, k) ∧ sel(m, i)
    endfor
end
```

Figure 7: *Coalescing two chained multiplexors.*

- If a multiplexor has multiple identical inputs, they are merged into a single input and the predicate is set to the logical "or" of the corresponding predicates
- A multiplexor with a single input is removed and the input is connected directly to the output.

Note that the result of this algorithm is not identical to Figure 4(C) (but it is equivalent). However, if we add a third multiplexor simplification rule (see Figure 7), which we have not yet implemented we obtain the same result. This third rule transforms two chained multiplexors (one being the unique output of the other) into a single multiplexor.

We have denoted by $sel(m, i)$ the predicate corresponding to the input i of multiplexor m.

3.6 Tokens

Logically, each data signal has an associated "token" signal, which is used to indicate when the value signal is valid (i.e., the computation that generates the value has terminated). This technique is used in the asynchronous circuit design of micropipelines [30]. The tokens can be implemented as 1-bit wires connecting the producer and consumers of a value; the tokens act as "enable" signals for the consumers. When the computation is inside a loop body, we need also an acknowledgement signal from the consumer to the producer, to indicate consumption of the data value; this indicates when the wires connecting the producer and the consumer can be reused.

We expect that in synchronous hardware implementations most of the token signals can be optimized away by statically scheduling the operations with known latency (more precisely, we can use a single token for all operations executed during the same clock cycle, and we can dispense with the acknowledgement altogether). Passing tokens will remain necessary between producers which have unpredictable latency (i.e., remote operations, like memory reads and procedure calls) and their consumers.

Tokens are used not only to signal that data values are ready, but also to preserve the original program order between instructions which have side effects. (Most of the previous work on data-flow machines [24, 31, 14] could dispense with this feature because it was handling functional languages.) For instance, two store instructions that have no data dependency between them cannot be reordered if they may update the same memory location.

4 Resources Required for ASH Implementations

In this Section we present a preliminary evaluation of the required resources for the complete implementation of programs in hardware. We analyze a set of programs from the Mediabench [17] and SpecInt95 [28] benchmark suites.

Resources: Table 1 displays the resources required for the complete implementation of these programs in hardware. We do not include in these numbers the standard library or the operating system kernel. We disabled inlining for collecting these numbers. All the values are static counts.

For some of the operations it is fairly easy to estimate the required hardware resources; we listed these under the heading "bits", and the values indicate the approximate number of bit-operations required to implement them. For remote operations (memory access, call/return), the implementation size can vary substantially, depending for instance on the nature of the interconnection network. We report for these just total counts.

The columns in Table 1 are:

LOC: lines of source code, including whitespace and comments, before preprocessing

SAMs: number of SAMs generated

fp: floating-point operations

memory: load and store operations

call/ret: call and return operations

predicates: boolean operations computing predicates (all of them are binary or unary, so each is one operation)

arithmetic: estimated number of operations necessary to implement the integer arithmetic operations (constant multipliers are strength-reduced to a few additions; non-constant multiplies and divisions are assumed to use n^2 bits, where n is the input width)

mux: number of bit operations in multiplexors (number of inputs * input size)

loop_regs: number of bits in loop registers

Comments: The raw computation resources required (the total of the "bits" columns) is below 2 million for all benchmarks except `mesa`, which is below 5 million. Even by today's standards, these are reasonably small (modern CPU cores already use more than 30 million transistors each, and state-of-the art reconfigurable fabrics already provide roughly 1-million bit-operations). By discounting the density disadvantage for the reconfigurable circuits, but extrapolating using Moore's law, within the next 10 years we have enough resources to implement each of these programs completely in hardware.

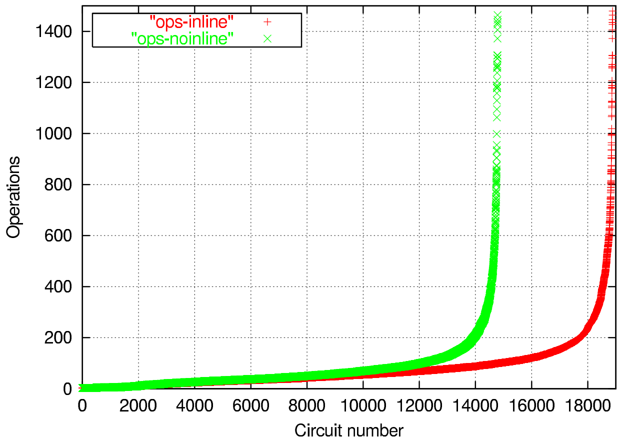


Figure 8: *Operations in each SAM, with and without procedure inlining. The data is for all the SAMs synthesized from all our benchmarks. The SAMs are sorted on their size and numbered in increasing order. On the x axis we have the SAM number and on the y axis the total number of "operations" per SAM. About 20 outliers above 1500 have been left out of the picture: the maximum size is 13100 without inlining and 14000 with inlining.*

This data doesn't include the savings that can be achieved by implementing computations of custom sizes. Research has shown [20, 6, 29] that static methods can eliminate a lot of the manipulated bits (methods developed by our own research [6] indicate that 20% of the bit computations in these benchmarks can be eliminated).

Notice that the resources taken by the predicate computations are minor compared to the actual computation; this suggests that large-granularity reconfigurable fabrics are more suitable for ASH systems than today's dominant style of FPGA, which has 1-bit functional units.

SAM/hyperblock size: Figure 8 shows a distribution of the SAM sizes, when measured in operations (this corresponds to the hyperblock size). If we use procedure inlining, we obtain more circuits, which tend to be larger. 90% of the SAMs use less than 200 operations.

Comments: Most SAMs are relatively small, which will translate into reduced power consumption and good locality for the intra-SAM signals.

ILP: In Figure 9 we plot the "average instruction-level

Benchmark	LOC	SAMs	Units fp	memory	call/ret	Bit-operations predicates	arithmetic	mux	loop_regs
adpcm_e	302	8	0	19	9	51	8,128	3,014	646
adpcm_d	302	8	0	19	9	51	6,144	3,014	646
g721_Q_e	1613	43	0	177	138	483	42,883	9,032	1,766
g721_Q_d	1619	41	0	180	137	486	42,837	8,804	1,635
gsm_e	6074	218	0	1,780	517	1,942	413,794	38,694	9,871
gsm_d	6070	214	0	1,768	513	1,936	413,014	38,436	9,805
epic_e	2701	312	75	498	295	1,335	300,665	69,848	29,689
epic_d	2452	231	36	719	242	1,125	230,418	66,922	28,021
mpeg2_e	7605	366	197	2,724	877	3,275	537,906	85,504	23,436
mpeg2_d	9832	316	11	1,789	839	2,624	305,803	45,570	13,333
jpeg_e	26881	1,331	153	8,693	1,964	8,167	1,022,023	200,354	76,722
jpeg_d	26115	1,285	153	8,248	1,889	7,625	996,243	194,374	75,897
pegwit_e	6713	270	0	2,112	608	1,346	151,160	24,039	7,857
pegwit_d	6713	270	0	2,112	608	1,346	151,160	24,039	7,857
mesa	65806	3,165	6,269	24,779	7,301	38,779	3,170,972	709,566	258,516
129.compress	1934	71	4	268	97	285	37,056	7,452	2,804
099.go	29246	1,778	0	9,610	2,966	19,350	1,309,148	367,116	104,987
130.li	7608	616	13	2,321	1,675	3,106	180,823	51,996	9,884
132.ijpeg	29265	1,427	163	8,556	2,321	8,141	1,101,735	202,652	78,630
134.perl	27072	1,406	48	14,348	4,997	34,363	1,377,612	467,864	99,015
147.vortex	67210	1,433	4	24,913	9,602	39,195	1,448,933	239,839	32,850

Table 1: *Static resource consumption for each benchmark. Some resources are expressed in units, while other are expressed in bit-operations.*

parallelism" in each SAM. We obtain this value by dividing the number of operations in a SAM (excluding inputs, outputs and constants) by the longest path within the SAM. The "average ILP" does not necessarily correspond to the dynamic ILP; the two values would be identical if all operations would execute with a latency of 1 clock cycle.

Comments: There are a few anomalous SAMs, which have an ILP of 0 or less than 1; one such SAM is SAM 1 in Figure 2 which contains no computations, so has a 0 count of useful operations. These SAMs can be eliminated by a special constant-propagation pass which we have not implemented yet.

We have isolated the ILP available just in the circuits that contain at least one feedback loop. These correspond directly to the program innermost loops, at least for the case of structured flow of control[10].

Fortunately, the ILP for the SAMs which contain loops is quite high: when we use inlining, 90% of the loops have an ILP above 2, about 50% have an ILP above 3, while 20% have an ILP above 5! In a sense (modulo the assumption about the identical latency of all operations), this ILP is the *sustained* ILP of these SAMs. These numbers enable us to believe that the performance of this model of computation can match or even exceed the performance

[10]Note that outer loops in the program map to several SAMs, so a SAM can be executed within a loop even if it contains no looping inside.

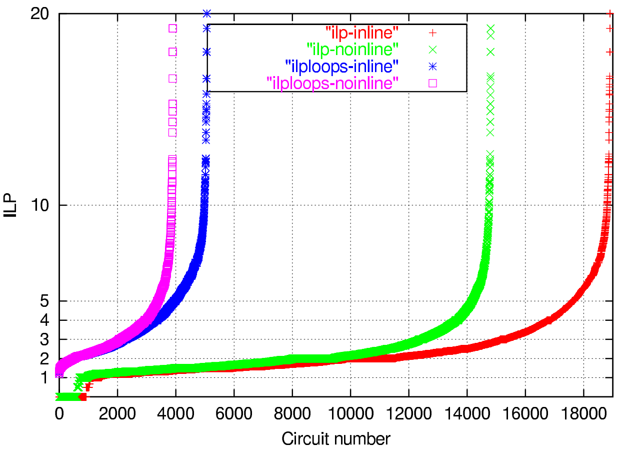


Figure 9: *Average ILP for each SAM, with and without inlining, and considering all SAMs or just the SAMs with loops. The data is for all the SAMs synthesized from all our benchmarks. The SAMs are sorted according to their ILP and numbered. The x axis is the SAM number. There are a few outliers above 20 not shown here: 4 SAMs have an ILP above 20, with a maximum of 41.55.*

of CPU-based systems.

Notice that we are being exceedingly conservative in scheduling the operations with side-effects: all memory operations and call/returns are enforced (using token-passing) to be made in the original program order, even if they are independent. We even impose a total ordering between memory reads! Once we remove this restriction, the ILP will definitely grow further. Notice that we have implemented common-subexpression and dead-code elimination, so the ILP is not artificially inflated by sub-optimal circuits.

5 Related Work

This work has two different lineages: research on intermediate program representation and compilation for reconfigurable hardware architectures.

Several researchers have addressed the problem of compiling high-level languages for reconfigurable and non-traditional architectures: [5, 22, 19, 7, 27, 26]. To our knowledge, our approach is unique in that it compiles very complex applications to hardware and it doesn't use a fixed number of computational resources.

The output of our compiler is a series of circuits. These bear a striking resemblance to some forms of intermediate representations of the program in other optimizing compilers. Our circuits are closely related to static-single assignment [12], dependence flow graphs [25] and value-dependence graphs [34], and, most closely, to gated-single assignment [32].

However, our circuits explicitly use predication and the notion of hyperblock; in that direction we are indebted to compilation for predicated execution machines: [21] and predicated static single assignment [9]. Unlike these program representations, we explicitly build the code to compute the predicates and we instantiate the ϕ functions through the use of multiplexors.

Our circuits are closely related to dataflow machines (see [33] for a survey), but our circuits are meant to be implemented directly in hardware and not interpreted on a dataflow machines using token-passing. The notion of Split-phase Abstract Machine is derived from the Threaded-Abstract Machine [11], a derivative of the dataflow work.

6 Conclusions

In this paper we have presented a proposal for a new model of computation, called Application-Specific Hardware (ASH), which implements programs completely in hardware, on top of a reconfigurable hardware platform. Our preliminary evaluations enable us to believe that soon we will have enough hardware resources to accommodate complete realistic programs, and that the sustained performance of this model will be comparable to processor-based computations.

We have discussed the compilation technology which can scalably translate large programs written in high-level languages into hardware implementations. Our compilation strategy transforms hyperblocks into circuits which execute many operations speculatively, and thus expose a substantial amount of instruction-level parallelism.

We have also outlined those features of the ASH model of computation that promise to provide *scalability* to this model: ASH implementations can easily and naturally take advantage of the exponentially increasing amount of hardware resources, avoiding many of the problems that the increased complexity brings to standard CMOS-based microprocessor design and manufacturing.

References

[1] V. Agarwal, H.S. Murukkathampoondi, S.W. Keckler, and D.C. Burger. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, June 2000.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 227–237, June 1998.

[3] David I. August, Wen mei W. Hwu, and Scott A. Mahlke. A Framework for Balancing Control Flow and Predication. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[4] Kaveh Azar. The History of Power Dissipation. *Electronics Cooling Magazine*, 6 (1), 2000.

[5] Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank Rajeev Barua, and Saman Amarasinghe. Parallelizing Applications into Silicon. In *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

[6] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations. In *Proceedings of the 2000 Europar Conference*, volume 1900 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[7] Timothy J. Callahan and John Wawrzynek. Instruction Level Parallelism for Reconfigurable Computing. In Hartenstein and Keevallik, editors, *FPL'98, Field-Programmable Logic and Applications, 8th International Workshop, Tallinn, Estonia*, volume 1482 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1998.

[8] Timothy J. Callahan and John Wawrzynek. Adapting Software Pipelining for Reconfigurable Computing. In *Pro-*

ceedings International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2000, 2000.

[9] L. Carter, E .Simon, B. Calder, L. Carter, and J. Ferrante. Path Analysis and Renaming for Predicated Instruction Scheduling. *International Journal of Parallel Programming, special issue*, 28(6), 2000.

[10] Katherine Compton and Scott Hauck. Configurable Computing: A Survey of Systems and Software. Technical report, Northwestern University, Dept. of ECE, 1999.

[11] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, July 1993.

[12] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[13] James R. Heath, Philip J. Kuekes, Gregory S. Snider, and R. Stanley Williams. A Defect-Tolerant Computer Architecture: Opportunities for Nanotechnology. *Science*, 280, 1998.

[14] Steven K. Heller. Efficient Lazy Data-Structures on a Dataflow Machine. Technical Report MIT-LCS-TR-438, Massachusetts Institute of Technology, 1989.

[15] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, second edition*. Morgan Kaufmann, 1996.

[16] R. Ho, K. Mai, and M. Horowitz. The Future of Wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[17] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Micro-30, 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, 1997.

[18] I. Lemke and G. Sander. Visualization of Compiler Graphs. Technical Report Design report D 3.12.1-1, USAAR-1025-visual, ESPRIT Project #5399 Compare, Universität des Saarlandes, 1993.

[19] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-Software Co-Design of Embedded Reconfigurable Architectures. In *DAC 2000*, 2000.

[20] S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth Sensitive Code Generation in a Custom Embedded Accelerator Design System. In *Proceedings of the 5th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2001)*, St. Goar, Germany, March 2001.

[21] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, Dec 1992.

[22] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proceeding of the International Conference on Computer Architecture 2000*, June 2000.

[23] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI00)*, 2000.

[24] Gregory Michael Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. Technical Report MIT/LCS/TR-432, Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[25] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence Flow Graphs: An Algebraic Approach to Program Dependencies. In SIGPLAN, editor, *In Principles of Programming Languages*, volume Volume 18, 1991.

[26] Rahul Razdan. *PRISC: Programmable reduced instruction set computers*. PhD thesis, Harvard University, May 1994.

[27] K. Sankaralingam, R. Nagarajan, D.C. Burger, and S.W. Keckler. A Technology-Scalable Architecture for Fast Clocks and High ILP. In *5th Workshop on the Interaction of Compilers and Computer Architecture*, January 2001.

[28] Standard Performance Evaluation Corp. *SPEC CPU95 Benchmark Suite*, 1995.

[29] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth Analysis with Application to Silicon Compilation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, 2000.

[30] Ivan Sutherland. Micropipelines: Turing award lecture. *Communications of the ACM*, 32 (6)(720–738), June 1989.

[31] Kenneth R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report MIT-LCS-TR-370, MIT, August 1986.

[32] Peng Tu and David Padua. Efficient Building and Placing of Gating Functions. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 47 – 55, 1995.

[33] Arthur H. Veen. Dataflow Machine Architecture. *ACM Computing Surveys*, 18 (4):365–396, 1986.

[34] D. Weise, R. F. Crew, M. Ernst, and B Steensgaard. Value Dependence Graphs: Representation Without Taxation. In *Proceedings of the Twentyfirst Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 297–310, January, 1994.

[35] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.

A Transducer Sensitive Task Allocation Algorithm for Distributed Embedded Systems

William Nace
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15217
wnace@cmu.edu

ABSTRACT

Distributed embedded systems impose an additional constraint upon the well known problem of task allocation. Because such systems interact with the physical world, sensor/actuator device interface software must be allocated to particular computing nodes based on the location of hardware interfaces. We have been able to exploit this constraint by creating an algorithm to pre-allocate tasks based on interface locations as a prelude to executing generalized task allocation algorithms. This paper describes an exploration of bin packing heuristics for this pre-allocation phase. A strategy based on packing tasks near interfaces first gives good packing quality and an overall task allocation speedup factor of 2.7 compared to previous approaches.

1. INTRODUCTION

Embedded systems, much like most other computing systems, are increasingly complex. In many systems, the complexity has been partitioned physically, resulting in highly distributed embedded systems. For instance, Modern automobiles consist of several networks and tens to hundreds of microcontrollers. Similar counts are commonplace in other transportation systems, factory automation, telecommunication and defense systems. The Robust Self-customizing Embedded Systems (RoSES) project is examining mechanisms to provide automatic graceful degradation to distributed embedded systems [11]. Our concept of operations involves the reconfiguration of fine-grained software components to the available hardware. Whenever a fault is detected, a reconfiguration manager will determine what hardware is still functioning, select mobile software components from a large library in order to maximize the functionality of the remaining system, and load the selected components onto the hardware. In order to determine the final location of the mobile components, the reconfiguration manager must execute a task allocation algorithm.

The task allocation algorithm to be used is a bit different from previously explored algorithms. In this case, the hardware is fixed and heterogeneous. Most other allocation algorithms come from the field of hardware/software co-design, where the hardware specification is part of the output of the problem, and is thus not fixed. The optimization sought in this case is usually a cost measure — silicon area, for instance. Parallel processing allocation algorithms also determine the mapping between software tasks and the homogeneous hardware processors that will execute the tasks. The goal of such algorithms is usually to minimize the schedule length of execution of the tasks. In contrast, distributed embedded systems are most frequently composed of many different microcontroller types, each with different amounts of compute resources, and very limited network bandwidth.

In addition to the minor differences caused by such fixed, heterogeneous hardware, the allocation desired by RoSES is constrained by an additional factor, unlike classical co-design or parallel processing realms. The software components of a distributed embedded system are managing and interacting with the sensors and actuators of the system. Those hardware components are not general — the fuel-air sensor in the automobile is located in a particular place, hooked to a particular microcontroller. It does no good for the allocation algorithm to think of moving the fuel- air sensor's driver software to any other microcontroller — it must be co-located with the sensor. Likewise, any software that interfaces directly to any hardware component is fixed — it cannot be allocated elsewhere.

The algorithm described herein exploits the fixed nature of hardware interface software components by examining the other software components (*tasks*) that it might call or be called by. These neighboring tasks can often be allocated locally and thus save any network communication. The process continues, allocating neighboring tasks in an attempt to minimize network usage. The remainder of this paper examines the details of how to choose the tasks to allocate, how many neighbors to examine and what to do when the processing elements with the transducers are filled. Section 2 takes a necessarily cursory look at the large body of related knowledge. Section 3 defines our system model. Section 4 discusses our core algorithm. Section 5 examines alternate policy choices. Results are summarized in Section 6. Finally, we present our conclusions and discuss future work in Section 7.

2. RELATED WORK

Task allocation is related to the well-known bin packing problem. In even a two-processor form, it is NP-complete [5]. Because of the applicability to OS scheduling on multi-processor systems, a great body of heuristic algorithms and analyses exists [8, 16, 14, 13, 2, 3, 6, 4]. See [9] for an excellent bibliography of such efforts. The two basic approaches to solving bin packing problems are list processing, where the objects are sorted and placed in the bins according to their order, and guided search, such as simulated annealing, where an initial solution is incrementally improved. The algorithm presented in this paper is a list processing algorithm.

In a multi-processor scheduling algorithm, the metric being optimized is generally the length of the critical path schedule. All parameter values used for the allocation are time units for each task to process or for communication to be transmitted. In contrast, a distributed embedded environment, the algorithmic interest is to ensure tasks can execute together on the limited resources of the microcontrollers.

The development of the transducer sensitive allocation algorithm is based to a large degree on the work of Beck[1]. Beck used a design advisor (DA) algorithm to generate a system hardware specification to meet the requirements of the software. The DA algorithm bin-packed vector valued software requirements (*i.e.* CPU cycles, RAM, ROM, I/O channels) into multi-dimensional bins representing the resources of the microcontrollers. Whenever the packing algorithm failed, the DA would expand the hardware specification. The basic idea is similar to much hardware/software co-design research — allocate software to the hardware to test if a partitioning decision is correct [7]. Prakash used linear programming techniques for a similar problem — simultaneous specification and allocation — though the application of such techniques to problems with large numbers of tasks appears to be computationally challenging[12].

The large size of this research area has spawned at least two attempts for standardization of the system descriptions. The use of standard task graphs facilitates benchmarking and comparison of the allocation algorithms. Kwok, *et. al.* collected 11 graphs from published papers (all of 7-18 vertices in size), combined them with a large number of randomly created graphs, and proceeded to benchmark the various scheduling algorithms[9]. Unfortunately, we have been unable to obtain this graph set for use in this research. The "Standard Task Graph" project has randomly generated a set of large graphs (30-2700 vertices) for the same purpose[15]. The standard graphs with communication costs, which would be most useful for our research, are not yet available.

3. SYSTEM MODEL

The software to be allocated is a collection of mobile components called *tasks*. The tasks are joined in a *task graph*, TG(V, E) whose vertices are the tasks and edges $E_{i,j}$ represent communication between task$_i$ and task$_j$. Edges may be directed, though doing so has no effect on this algorithm. Each task$_{i,j}$ is labeled with its processing requirements, *p(i)*. Processing requirements are often a list of multiple independent values such as CPU cycles, RAM, or I/O

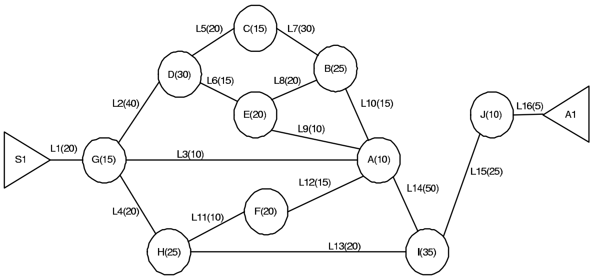


Figure 1: A sample task graph

channels. Similarly, edges are labeled with communication requirements *c(i)*, usually representing bandwidth. Figure 1 shows a sample task graph. This particular graph, from [4], is often referenced in the allocation and task scheduling research field. *p(i)* and *c(i)* are shown in parenthesis.

Originating vertices (those with no inputs) represent *sensor* components that are the source of data. Likewise, those vertices with no outputs are *actuators* which act as a sink of data. We use the term *transducer* to describe both sensors and actuators. The task graph in Figure 1 has a single sensor (S1) and actuator (A1).

The components of the task graph must be mapped to the available hardware. The hardware is a collection of *processing elements* (PEs) connected to a single network. Each PE has a fixed resource list, in the same size and types as the requirements of the tasks. Likewise, the network resources match the requirements of the communication edges. Additionally, transducer tasks are pre- assigned to PEs, as their physical hardware is not general to all PEs. This system model uses a single network, though an extension to multiple networks is also possible. The general approach for such an extension would follow the techniques described in [10].

The major opportunity for optimization occurs by allocating incident tasks to the same microcontroller. In such a case, the communication requirement *c(j)* is fulfilled through local intertask communication, and thus does not impact the network at all. In contrast, if the tasks are allocated to different microcontrollers, *c(j)* must be borne by the network.

Because we are not overly concerned with schedule, but rather resource usage, we make the assumption that the program will cycle continually. This assumption is quite reasonable for embedded systems where new samples are periodically available at the sensors and serviced in a time-triggered manner.

4. THE TRANS_FIRST ALGORITHM

The key insight behind the TRANS_FIRST algorithm is to exploit knowledge about transducer location. This bit of knowledge is not available when allocation algorithms are used for general purpose computing systems, as there is little reason to require a particular task to execute on a particular PE. But in a distributed embedded system, the transducer tasks are managing special purpose hardware only available at a particular PE, so they must be allocated to those PEs.

Table 1: Policy Choices

Task Choice	PE Choice	Task at a Time	PE Fill Level
Largest (L)	Id Order (1)	Yes (Y)	Full (F)
Smallest (S)	Reverse Id (2)	No (N)	80% (8)
Max B/W (B)	Largest (L)		50% (5)
Min Neighbors (N)	Smallest (S)		
Max Neighbors (M)	Random (R)		
Random (R)			

By working inwards from the exterior vertices of the task graph, large sections, or subgraphs, of the graph may be allocated so as to eliminate network communication among tasks in the subgraph.

The basic algorithm is:

```
BEGIN {TRANS_FIRST}
  Chose a PE: pe                                    (5.2)
  REPEAT
    Initialize set T with all tasks which:
      Are incident to a task allocated on pe
      Can fit on pe                                 (5.4)
      Haven't been allocated yet

    REPEAT
      Select task: k from T                         (5.1)
      Remove k from T
      IF k can fit on pe THEN
        Allocate k to pe
        Add to T all tasks which:
          Are incident to k
          Can fit on pe                             (5.4)
          Are unallocated
          Haven't been rejected before
      ELSE
        Remember that k has been rejected
    UNTIL (T is empty)
  UNTIL (All PEs considered)
  Allocate remaining tasks
END {TRANS_FIRST}
```

This algorithm is a variant of the list-processing heuristics for solving bin packing problems. Its usefulness and performance will depend upon the particular policies for making decisions within the algorithm. We examine the alternate policy choices available at the numbered lines in the corresponding portion of the following section.

5. POLICY CHOICES

The TRANS_FIRST algorithm is affected by four basic policy choices, three of which were illustrated in the algorithm pseudo-code of Section 4. The fourth (Task at a Time) requires a slight re-organization of the algorithm and will be described fully in Section 5.3. All of the choices are listed in Table 1, along with an identifying character for easy reference in the results tables.

5.1 Choosing a Task for Allocation

This is the policy choice with the most latitude. Tasks can be chosen from the set of candidates based on the characteristics of the particular task, or of their neighborhood of the task graph. We chose to examine six alternatives. The use of task size follows from the well-known bin packing rule-of-thumb: "pack the largest item first." We contrast this approach by also attempting to pack the smallest first. Often, the network bandwidth is a scarce resource worth conserving. To that end, we choose tasks based on the bandwidth savings offered by a local allocation. Recall that communicating co-located tasks need no network resources.

In an attempt to choose tasks based on their neighborhood in the task graph, we examine the results of a choice based on the numbers of neighbors. By choosing tasks with the most neighbors, we provide a bigger pool for choices in successive iterations. The danger, however, is that the subgraph will consist of several tendrils that block off and interfere with the growth of other subgraphs, without making the kind of bundles that can really pay off from a network perspective. We also consider a policy which chooses the task with the lower number of neighbors. Such a policy should consume all the tasks in a region of the graph before expanding.

For comparison purposes, we also randomly choose tasks from a uniform distribution. The random choice uses absolutely no knowledge of the graph or task charactersitics, so provides a baseline which the algorithm must be able to outperform in order to be of any use whatsoever.

5.2 Choosing a PE

Clearly if the subgraphs allocated to the various PEs are not adjacent, then the order in which PEs are chosen for allocation will have no effect on the solution. Conflict occurs only when some task has the potential for allocation to multiple PEs. One would think that on large task graphs such potential would be rare. Such intuition is incorrect for the distributed embedded task graphs, as the transducers cluster in particular neighborhoods of the graph.

To explore the degree to which PE choice policy affects the solution, we implement 5 different alternatives. The first uses an arbitrary, though fixed, ordering — by identification number. We also examine the reverse ordering. The only characteristic of the PE which may have some bearing on the potential solution is the amount of resources the PE has available. The PE with a large resource stash should be able to carve out a larger subgraph if unimpeded by other PEs. Such a policy choice is not clearly a winner, as such a large subgraph may block off the later PEs from access to any portion of the graph. For this reason we study PE ordering by resource in both ascending and descending order.

We also include random choice as a baseline, for the same reasons as in Section 5.1.

5.3 Task at a Time Allocation

The TRANS_FIRST algorithm, as described in Section 4, allocates all the tasks that fit on a PE before allocating any to other PEs. In a task graph where many sensors are coupled via a few tasks, such a strategy will generate a large subgraph for the first PE chosen. But the subgraph

Table 2: Task Graph Characteristics

Graph	Tasks	Edges	Sensors	Actu-ators	PEs
ranA	80	200	8	10	9
ranB	80	200	8	10	5
ranC	42	110	1	6	6
ranD	43	100	13	13	9
ranE	17	50	22	21	11
ranF	20	50	4	1	10
iac	14	98	3	64	9
tract	43	282	2	115	16
sch	34	316	3	117	16

will block off development of subgraphs on closely associated PEs. By re-organizing the main loop of the algorithm, a "breadth-first" strategy can be attempted, where a single task is allocated from a PE at any particular time. The re-organized algorithm must maintain the state of each PE's search in independent T sets. After a task is allocated from one PE, the algorithm queries another PE, in the same order specified for Section 5.2.

5.4 PE Fill Level

By allowing PE resources to be fully consumed by this pre-process step, allocation of other large tasks may be inhibited. It is possible that restricting task allocation during the first phase of TRANS_FIRST may conserve space for the tasks at the middle of the graph that are far from transducers. We explore allocating tasks only to fill 80% (or 50%, or some other arbitrary cap) of PE resource levels. The inevitable tradeoff is that packing to 80% on all the PEs may then leave us vulnerable to a 21% (or 51%) sized task. This tradeoff is merely another case of the typical best-fit versus worst-fit bin packing policy choice.

5.5 Allocation of Remaining Tasks

Once all PEs which host transducer tasks have been filled using the TRANS_FIRST algorithm, remaining vertices of the task graph may remain. Use of another bin packing algorithm will allocate them to the remaining PEs.

Typical distributed embedded systems will not have many (or any) PEs remaining at this point. Rather, all PEs are connected to, or encompassing on the same silicon, the system's sensors and actuators. This "smart sensor" strategy leaves few PEs remaining as merely compute nodes. In the case of a transducer hardware failure, however, the micro-controller is still capable of operation and would be allocated tasks from the central portion of the task graph.

6. RESULTS

6.1 Test Set

All experimentation was done using a collection of 9 graphs from [1]. Six are randomly generated graphs and 3 are from real-world automotive applications. The salient features of each graph are shown in Table 2. PE and network resource sizes were developed via execution of the system specification generation algorithm from [1].

6.2 Experimental Method

We first present four experiments to determine the appropriate policy generation choices. An additional experiment compares the TRANS_FIRST algorithm to system allocations done without sensitivity to transducer location. Both algorithms were implemented using as much of the same code as possible (to screen out implementation differences) and all executions done on the same computer (A 750MHz Pentium 3, in an IBM Thinkpad T20). Each experiment involved 10 runs of each algorithm choice. Averaged values over the 10 runs are reported.

Three values of interest were measured and calculated for each of the graphs: success rate, network usage and algorithm running time. Success rate is a measure of the number of times the algorithm found an allocation over repeated execution of the algorithm. Unless explicitly using a random policy choice, the TRANS_FIRST algorithm is completely deterministic – though the follow on phase, as described in section 5.5, is not. Success rate, therefore, is often 100% or 0%, regardless of the number of executions.

Network usage is a figure of merit which measures the extent to which an allocation placed edges of the task graph in locally. It is calculated as the ratio of the bandwidth of the task graph edges with incident tasks placed on the same PE to the total bandwidth of all task graph edges. A value of zero indicates that all communication is on the network, while a value of one is the (highly unlikely) case where all communication is local to a PE.

6.3 Finding the Right Policy Choices

6.3.1 *Task Choice*

Table 3 shows the results of 6 experiments, each differing in the task selection policy. Choosing the task with the largest requirement is the best of the policy choices. Note that no other policy choice had a higher success rate on any of the graphs. It is less impressive with respect to network usage — a situation that is not surprising given that such a policy choice pays no attention to the network at all. All of the graph sensitive choices (Min Neighbor, Max Neighbor and Bandwidth Savings) have good network usage statistics. Those choices do not react to the resource requirements for actual packing on the PE, so they do not actually allocate with good success rates. We use the Largest Task policy choice for all other experiments.

6.3.2 *PE Choice*

Table 3 also shows the results of the 5 experiments on PE choice. Using an arbitrary ordering (By ID and By Reverse ID) shows no difference in packing success, but does use the network to a strikingly different degree most notably in the real-world task graphs. Both are still outperformed by a Random choice and both resource level choices. Selecting the Largest Resource policy is a narrow winner over Random with respect to packing success. However, Largest Resource allocates many more communication edges to local PE communication. Our hypothesis in this regard is correct: choice of a large PE allows for large subgraphs to be allocated locally, without restricting further development from the smaller PEs. We use the Largest PE policy for the following experiments.

Table 3: Experimental Results

Experiment 1: Task Choice

	Network Usage									Success Rate								
Policy	ranA	ranB	ranC	ranD	ranE	ranF	sch	tract	iac	ranA	ranB	ranC	ranD	ranE	ranF	sch	tract	iac
L	.35	.53	.35	.45	.70	0	.17	.21	.23	.8	.8	.9	1.0	1.0	0	.8	.8	1.0
S	0	0	.42	0	.70	0	.17	.24	.38	0	0	.2	0	1.0	0	.1	.6	.9
B	.44	0	.56	0	.68	0	.22	.27	.29	.2	0	.6	0	1.0	0	.7	.7	.9
N	0	.48	0	.46	.70	0	.16	.29	.33	0	.4	0	.9	1.0	0	.8	.4	1.0
M	.32	.44	.41	.44	.70	0	.18	.29	.31	.3	.4	1.0	.1	1.0	0	.7	.7	1.0
R	.40	.45	.38	.46	.69	0	.18	.24	.29	.2	.3	.8	.3	1.0	0	.6	.6	1.0

Experiment 2: PE Choice

2	.37	0	0	.48	.68	0	.16	.20	.27	1.0	0	0	1.0	1.0	0	1.0	1.0	1.0
1	.33	0	0	.48	.72	0	.41	.49	.71	1.0	0	0	1.0	1.0	0	1.0	1.0	1.0
L	.32	.59	.35	.48	.73	0	.40	.26	.35	1.0	.7	1.0	1.0	1.0	0	1.0	1.0	1.0
S	0	.45	.43	.44	.68	0	.21	.26	.22	0	1.0	1.0	1.0	1.0	0	.5	1.0	1.0
R	.35	.53	.35	.45	.70	0	.17	.21	.23	.8	.8	.9	1.0	1.0	0	.8	.8	1.0

Experiment 3: Task at a Time

Y	0	0	0	.47	.67	0	0	.24	.20	0	0	0	1.0	1.0	0	0	1.0	1.0
N	.32	.59	.35	.48	.73	0	.40	.26	.35	1.0	.7	1.0	1.0	1.0	0	1.0	1.0	1.0

Experiment 4: PE Fill Level

F	.32	.59	.35	.48	.73	0	.40	.26	.35	1.0	.7	1.0	1.0	1.0	0	1.0	1.0	1.0
8	0	.59	.29	.42	.68	0	.33	0	0	0	1.0	1.0	1.0	1.0	0	1.0	0	0
5	.29	.59	.41	.41	.70	.20	0	0	0	1.0	.8	1.0	1.0	1.0	.1	0	0	0

6.3.3 Task at a Time Allocation

In Section 5, we advanced the supposition that it may be better to allocate a single task from the PE before trying a different PE. The experiment documented in Table 3 shows this not to be the case. The Task at a Time policy resulted in successful allocations on only 4 of the 9 graphs. It turns out that this allocation policy fills up each PE somewhat equally, without a reserve in case a large task is encountered. We use the PE at a Time allocation policy.

6.3.4 PE Fill Level

One remarkable aspect of the results shown in Table 3 is the complete lack of success for any policy on the ranF graph. In this particular case, a large task is near the center of the graph, and thus out of reach of each of the PEs until they have already allocated some tasks. Unfortunately, by filling up with smaller tasks, the PEs have no remaining resources for the large task. In our final experiment, we attempt to limit the task allocation during the transducer sensitive phase of the algorithm in order to save some space for such large tasks. Our experiments show this approach is generally unsuccessful. However, by leaving a 50% cap in place, we have our only success — a limited one — with the ranF task graph. We speculate that an adaptive cap, based on the sizes of tasks actually occurring in the task graph, may be more feasible. Such exploration will await further effort. We will conclude that the best general set of policy choices is L-L-N-F (Largest Task Size, Largest PE first, PE at a Time, Full PE).

6.4 Comparison to Base Algorithm

A careful examination of the Beck algorithm[1] reveals a few strengths of the TRANS_FIRST algorithm: complete determinism and improved execution time. Table 4 shows a comparison. BECK is almost as good terms of packing success — it is, after all, a very good algorithm. However, BECK has a surprisingly large random component. Table 4 does not show this effect, but the values that were averaged to get net usage vary quite a bit. BECK orders the tasks by their size and packs in decreasing order to the PE which would minimize the network bandwidth. Early in the algorithm's execution there are quite a few ties, where placement to any PE would take zero bandwidth (since the task's neighbors haven't been allocated yet). Such ties are resolved randomly, which significantly affects the remainder of the execution. The policy choices we've selected for TRANS_FIRST preclude any random elements, thus reserving any randomness for the follow on allocation. Randomness is not, of course, always a bad thing. If BECK fails to find an allocation, it is always possible to re-execute it to see if it will find a different, successful, allocation.

Table 4 also shows the TRANS_FIRST algorithm has a clear time advantage. The average speedup of 2.7 is substantial, and is a result of the "divide and conquer" nature of the algorithm. By operating on small portions of the entire task graph (the regions near the transducers), choices are made among a much smaller set of tasks. Such small comparisons are much quicker than the large comparisons required by BECK as it examines the entire graph.

Table 4: Comparison to Base Algorithm

TRANS_FIRST

	ranA	ranB	ranC	ranD	ranE	ranF	sch	tract	iac
Net Usage	.32	.59	.35	.48	.73	0	.40	.26	.35
Pass Rate	1	.7	1	1	1	0	1	1	1
Execution Time (mS)	617	609	379	365	192	0	710	678	287
Speedup	2.2	1.8	2.3	3.2	3.4	0	3.1	3.1	2.8

BECK

Net Usage	.19	.62	.27	.39	.69	0	.43	.49	.6
Pass Rate	1	.6	1	1	1	0	1	1	1
Execution Time (mS)	1350	1080	855	1170	647	0	2215	2108	815

7. CONCLUSIONS AND FUTURE WORK

We have shown a task allocation algorithm that successfully exploits a constraint unique to the distributed embedded system domain. The algorithm works by allocating tasks that manage transducer hardware (and thus cannot be allocated elsewhere) to the local processing element, and then operating on the neighboring set of tasks. Large subgraphs are thus swept into a single processing element, which saves significant network bandwidth.

In order to tune the heuristic algorithm, a set of experiments was conducted. Each policy choice was clearly delineated and executed on a series of random and real-world task graphs. The following policy choices resulted in a heuristic algorithm with good packing quality and a substantial speedup: Largest Task First, Largest PE first, PE at a Time, and 100% PE Fill level.

In the future, we would like to examine the quality of the heuristic on additional task graphs; in particular, the sets of task graphs proposed by [15] and [9] as standard comparison graphs. In addition, to fit this algorithm into the RoSES graceful degradation research, we wish to determine if environmental conditions (*i.e.* the nature of the task graph and hardware specification) can be measured and then policy choices tuned to the observed conditions.

8. ACKNOWLEDGMENTS

We are quite grateful for the support of the General Motors Satellite Research Lab at Carnegie Mellon University and Bosch Electronics.

9. REFERENCES

[1] J. Beck. *Automated Processor Specification and Task Allocation Methods for Embedded Multicomputer Systems.* PhD thesis, Carnegie Mellon University, April 1995.

[2] S. Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Transactions on Software Engineering*, SE-7(6):583–9, Nov 1981.

[3] S. Bokhari. Partitioning problems in parallel pipelined and distributed computing. *IEEE Transactions on Computing*, 37(1):48–57, Jan 1988.

[4] K. Efe. Heuristic models of task assignment scheduling in distributed systems. *Computer*, 15(6):50–6, June 1982.

[5] M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness.* Freeman, San Francisco, 1979.

[6] B. Indurkhya, H. Stone, and L. Xi-Cheng. Optimal partitioning of randomly generated distributed programs. *IEEE Transactions on Software Engineering*, SE-12(3):483–495, Mar 1986.

[7] A. Kalavade and E. Lee. A hardware-software codesign methodology for dsp applications. *IEEE Design and Test of Computers*, 10(3):16–28, September 1993.

[8] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C33(11):1023–9, Nov 1984.

[9] Y. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, Dec 1999.

[10] G. McNally. *Automated Architecture Specification for Embedded Multicomputer Systems.* PhD thesis, Carnegie Mellon University, 1998.

[11] W. Nace and P. Koopman. A product family approach to graceful degradation. In *Architecture and Design of Distributed Embedded Systems*, Oct 2000.

[12] S. Prakash and A. Parker. SOS: Synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distributed Computing*, 16(4):338–51, Dec 1992.

[13] C. Shen and W. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a mini-max criterion. *IEEE Transactions on Computers*, C34(3):197–203, Mar 1985.

[14] H. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, Jan 1977.

[15] T. Tobita, M. Kouda, and H. Kasahara. Performance evaluation of minimum execution time multiprocessor scheduling algorithms using standard task graph set. pages 745–751, Jun 2000.

[16] C. Woodside and G. G. Monforton. Fast allocation of processes in distributed and parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):164–74, 1993.

The Design of a Secure Location System

Urs Hengartner*
Department of Computer Science
Carnegie Mellon University
uhengart+@cs.cmu.edu

Abstract

Ubiquitous computing poses new challenges on controlling access to services provided in such an environment. A key service is a system for locating people and learning about people in a particular room. Since location is a sensitive piece of information, the access control requirements of such a location system are rather stringent. We examine the threats a location system faces and present the design of a distributed solution that comprehensively addresses all of these issues. Our system exploits three basic concepts: trust, confidence values, and delegation. We rely on trust for dealing with misbehaving services. Confidence values help cope with location information of limited accuracy. Using delegation, entities in the system can have other entities make policy decisions for them. We demonstrate feasibility of our design with an example implementation of a secure location system.

1 Introduction

Ubiquitous computing environments such as the ones examined in CMU's Aura project [1] rely on the availability of location information about people. With the help of this information, location-specific services can be provided to a person in the system. However, location is a sensitive piece of information that should not be distributed to anyone.

This paper discusses the design of a secure location system for the Aura project. The two main characteristics of this design are its support for a variety of location technologies and its security mechanisms. Our system employs location technologies that are based on people carrying a badge or some other device with them, but also techniques not relying on devices, such as detecting who is logged in at the terminal of a computer. In terms of security, our systems is able to implement policies that specify who is allowed to get what kind of location information about someone. We make sure that location information is not forwarded to users that are not authorized to get this information. In addition, policies also allow to specify the kind of delivered location information, whereas kind is determined by the granularity of the delivered information (e.g., CMU Campus vs. Wean Hall 8220) and which location services are used to answer a query.

Possible location queries people might ask are 'Where is Alice', 'Where am I', 'Who is in/near Wean Hall 8220', 'How far apart are Alice and Bob', 'Is Alice within n feet of or in the same room as Bob', and 'Who is within n feet of Alice'. Further investigation of these queries reveals that all of them can be answered based on two native queries: 'Where is a particular person' ("user query") and 'Who is in a particular room' ("room query"). Therefore, we introduce two basic services: a "People Locator" service answering the first kind of queries and a "Room Locator" service replying to the second type of queries.

We assume a hierarchical model for our location system. An example of such a system, as it could be deployed in CMU's School of Computer Science (SCS), is given in Figure 1. The nodes in the graph are either services or devices, the arrows denote which service contacts which other service/device. The *location system* is a composition of multiple *location services*. Each location service either exploits a particular technology to gather location information or processes location information received from other location services. A user who wants to find out about the location of some other user contacts the People Locator service. Similarly, if he wants to find out who is in a particular room, he contacts the Room Locator service. The People and Room Locator service then contact other services which themselves may also contact other services and so

*Urs Hengartner has been advised by Peter Steenkiste.

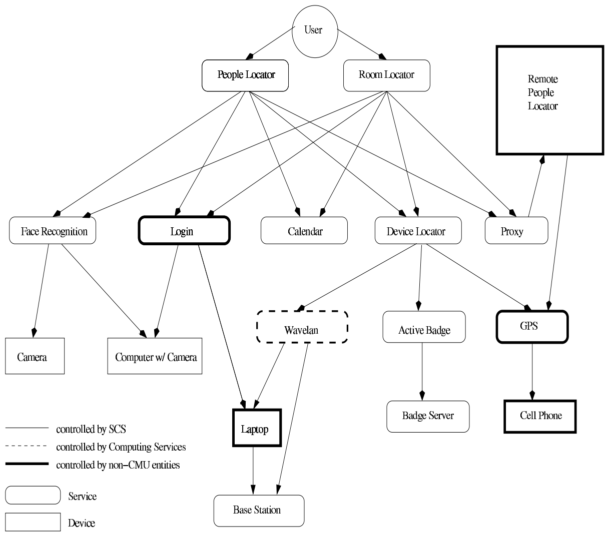


Figure 1: Example location service.

on. Location information flows in the reverse direction of a request (not shown in the figure).

In our example, two kinds of services are contacted. The first group consists of services that are aware of users. The face recognition, login, and calendar service belong to this group. The face recognition service tries to detect the location of users by capturing their image when they pass fixed cameras. The login service keeps track of which users are logged in at the consoles of computers. The calendar service delivers location information based on people's schedule. The second group of services are not aware of users, but of devices a user is carrying with her. These services are contacted by the Device Locator service. In our system, this group of services consists of the Wavelan, active badge, and GPS service. The Wavelan service keeps track of the location of wireless devices such as laptops. The active badge service is based on fixed sensors that receive RF or IR signals emitted by active badges. The GPS service calls GPS-enhanced mobile phones and gets their geographical location from them. Finally, if a user cannot be located by any of the services in the local location system, a proxy may contact remote location systems.

A basic assumption in our work is that the services outlined above are administrated by different entities. For example, the calendar service is controlled by SCS's computing facilities, the Wavelan service by CMU's computing facilities, the GPS service by a phone company, and the Login service by a single person.

The model in Figure 1 gives an overview of the services in our system and their interaction, but it does not show how these services are actually implemented. For example, the People Locator service can be implemented on a single node, on multiple nodes to improve scalability and robustness or even in a completely distributed way, where each user has her own People Locator service answering queries about her location.

The contribution of our work is a comprehensive analysis of the security requirements of a location system such as the one discussed above. We also provide the design of a solution for the outlined issues and present an example implementation of this design.

The outline for the rest of this paper looks as follows: In Section 2, we elaborate on user and room location policies. The threats a location system faces are listed in Section 3. We present a strawman solution and its weaknesses in Section 4. The actual design of our solution is given in Section 5. It relies on digital certificates, which are discussed in Section 6. Section 7 describes our implementation. We elaborate on related work in Section 8 and on our conclusions and future work in Section 9.

2 Location Policies

Location information about a person or about the people in a room should be forwarded to someone only if allowed by that person's and that room's location policy, respectively. In this section, we discuss such location policies in more detail. This discussion will reveal some of the requirements that need to be fulfilled by our location system, more specifically, by the mechanisms we exploit to implement this system.

2.1 User and Room Location Policies

We assume that there are two kinds of users: located users and locating users. Locating users query the location system for a located user.

Our system knows about two kinds of location policies: user location policies and room location polices. In a user location policy, a located user states who is allowed to get what kind of location information about her. In a room location policy, the "owner" of a room states who is allowed to find out about the people currently in this room and at what granularity level (e.g., Alice vs 'someone'). The owner of a room is typically the institution or the company the room belongs to. However, the institution might decide to give owner rights and thus the right to decide about the room location policy to other people. For example, in the case of an office, its occupant can be declared to be its owner. For meeting or lecture rooms, the institution would probably keep the owner rights (or give only a subset of them away, for example, to find out whether anyone is in a meeting room or whether it is empty).

2.2 Conflicting Policies

Each located user has a user location policy stating who is allowed to locate her. Similarly, for each room, there is a room location policy stating who is allowed to find out about the people in this room. If these two kinds of policies are established independently of each other, access control conflicts might arise. For example, Alice might not allow Bob to locate her, on the other hand, Charles might allow Bob to locate people in his office, so assuming Alice is in Charles's office, what should the result of a 'Who is in Charles's office' query asked by Bob be?

A conservative approach is not to return any information about Alice if either of the two policies does not allow this information sharing. While this is a feasible approach, we argue that it is too conservative for our purposes. Imagine an Aura service that warns people when someone enters their office so that, for example, sensitive information projected onto a wall is hidden automatically. If the entering user sets her location policy to reveal no information, this service will break. It is our belief that the owner of a room should always be allowed to find out who is in his room, regardless of the location policies of the users currently in his room. Similarly, if a user states that someone is allowed to locate her, then this lookup should always succeed, regardless of the location policy of the room she currently is in. Therefore, in our system, access control for user queries is based only on the user location policy, similarly, for room queries, the system looks only at the room location policy.

It is possible to combine the room and user location policy. The owner of a room can specify in his room location policy that information about a person in his room should be released to a locating user only if this person grants the locating user access to her location information in her user location policy. We envision that such a room location policy is most appropriate for lecture or meeting rooms.

2.3 Transitivity of Access Rights

Another interesting question is transitivity of access rights. Assuming Alice grants Bob access to her location information, should Bob be allowed to forward this access right to Charles? A similar question arises for room location policies: If Ed is allowed to find out about the people in his office, should he be allowed to give this access right to Fred? There is no final answer for either of these questions. For the first case, it should probably be Alice's decision whether this forwarding should be allowed. For the second case, we argue that in most cases, the answer should be no. If Ed is given the option to let other people find out about Alice's location when she is in his office, Alice's location policy might be violated. In addition, Ed could give someone not within the same institution access rights to his office, which might not be in the institution's interest.

Because of the reasons outlined above, we argue that our location system should let located users/owners of rooms explicitly state whether they want transitivity of access rights.

However, note that even though a user might not be allowed to forward his access rights to other users, he could still proxy for them. In the example above, assume that Alice does not give Bob the right to forward his access rights. Nonetheless, Bob could still serve as a proxy for Charles and deliver location information about Alice to him through channels our location system is not aware of. The only way to handle such an information leak is to take away the access rights from Bob completely.

3 Threats

In this section, we discuss the threats a location system such as the one introduced in Section 1 faces. A threat is identical to having a service misbehave. We call a service 'misbehaving' if it suffers from at least one of the following problems:

Wrong location information. Each service or device in the calling chain starting at the People or Room Locator service and ending at a helper service or at a device may return wrong location information. Services such as the Wavelan service that generate their own location information may generate wrong information, services such as the Device Locator service that forward information received from other services may falsify received information.

Unauthorized location information. A service might return location information to unauthorized users. Located users do not want every locating user to be able to get information about their location. In addition, they may be willing to release information about their location to some users, but only of limited accuracy. A located user's preferences are specified in her location policy. Our system has to guarantee a located user that her policy is implemented correctly. It should not be possible for a locating user to get unauthorized location information, neither by contacting the People Locator service nor by directly contacting one of the helper services. A similar guarantee has to hold for room location policies.

Wrongly contacted services. A service such as the People or Device Locator service may contact other services that the located user does not want to be contacted. Although our location system is able to exploit multiple location technologies, a located user does not necessarily want the People and Device Locator service to contact all available services. For example, a service may charge the located user for each request (e.g., the GPS-based location service), thus the located user allows only a subset of the locating users to get location information from this service.

Of course, we would like our location system to eliminate the threats mentioned above. In Section 5, we describe in more detail how our solution addresses them. However, note that it is not feasible to completely eliminate the threats due to the following reasons:

Uncontrollable systems/devices. We cannot prevent all hosts covered by our system from forwarding location information to random people. For example, it is up to the owner of a workstation to set up access control to the finger daemon running on his workstation. Potentially, anyone can get access to the finger and login information (and thus potentially location information). Even if all hosts were secured, there might still be ways for random people to get location information. There may be services not in our system that allow to retrieve location information about someone. For example, some people put their schedule into a .plan file that is accessible by anyone. In addition, as soon as a person uses services like Wavelan, her location is known to at least the operator of the Wavelan service. If she does not trust this operator to deal with her location information sensibly, her only alternative is not to use Wavelan.

Wrong source information. Even when we assume that all hosts/services in our system do not actively generate wrong information, there are still ways for the system to return wrong location information. For example, the face recognition service could easily be tricked by presenting a camera someone's picture. All the login service is able to detect is that someone with a particular user account has recently been active at a console, there is no way for it to verify that the user owning this account is actually sitting in front of the computer or whether she has allowed her friend to use her computer. For device-based services, there are similar problems. If you lend your laptop to a friend without telling the location system, the system will either deliver wrong location information or it may have to deal with conflicting information in case your location can also be detected by some other means. In our location system, we assume that users are willing to cooperate with a certain set of services and thus do not try to fool these services. We elaborate on this concept in Section 5.1.

Inaccurate information. Some location information is inherently associated with a certain level of inaccuracy. For example, the longer a user logged in at the terminal of a computer has been inactive, the bigger becomes the probability that she is no longer sitting in front of her computer. Therefore, a location system needs to be able to deal with location information that might be out of date or inaccurate. Our solution to this problem is outlined in Section 5.2.

4 Strawman Design

In this section, we present a strawman design of the people location system. We also discuss how it handles the threats mentioned in Section 3. Our discussion concentrates on user queries, but room queries could be handled in a similar way.

In this system, we keep a policy matrix at a centralized, trusted server that lists for each locating and located user pair what the access control policy is. Each location service is required to clear requests with the centralized server before executing them. Alternatively, to reduce overhead, only the first service in the calling chain, the People Locator service, contacts the centralized server and has it approve the request in a way visible to the other services in the chain by, for example, signing the request with a digital signature.

This design suffers from several shortcomings:

Bottleneck. Having each request being approved by a centralized server lets this node become the bottleneck of the system.

Misbehaving services. The system fails to address one of the threats mentioned in Section 3; it does not provide a way for users to deal with misbehaving services that return wrong or unauthorized location information.

Unknown users. Policy matrices assume that the location system knows all the located and locating users. Whereas this assumption makes sense for located users (else there is no way to locate them), it does not need to hold for locating users. Anyone who the located user wants to learn about her location should be able to do so, regardless of whether he is known to the system.

Group access. Very often, users want to give an entire group of people (e.g., all their friends) the same kind of access rights. Policy matrices make handling groups tedious and error-prone since there is no way to directly give/deny access to a group of people. Instead, each user has to be given/denied access separately.

To conclude, having a centralized server maintain a policy matrix is not flexible enough. In the next section, we propose a flexible solution that succeeds in resolving the issues mentioned above.

5 Design of the Secure Location System

Based on our discussion in Sections 3 and 4, we now present the design of our secure location system. The main concepts we exploit in our solution are trust, confidence values, and delegation. First, we give users the opportunity to specify which services they trust. This trust can be extended to entire organizations/administrative entities. Second, we associate location information with a confidence value reflecting its accuracy. Third, we allow services to delegate particular rights to other services and people.

In the following sections, we elaborate on these three concepts and how they are exploited in our solution in more detail. For implementation purposes, we assume that there is some kind of a "digital ticket" that states a trust or delegation decision and that is signed by the located user or a service. Digital tickets can be implemented using, for example, Keynote [4] or SPKI/SDSI [5] certificates. An actual implementation based on SPKI/SDSI is discussed in Section 6. Confidence values are probability values between zero and one.

5.1 Trust

Our solution for dealing with misbehaving services relies on trust. We assume that the located user or the owner of a room trust at least some of the services in our system. The trust assumptions are

- that the service implements a location policy faithfully when it is given such a policy,
- that it does not falsify information when it has to forward or process location information retrieved from other location services,
- that it does not deliberately generate and return wrong information when it generates its own location information[1],
- that it does not contact services the located user does not want to be contacted, and
- that it does not forward location information to untrusted services.

[1] A service can undeliberately generate wrong location information, as pointed out in Section 3.

A service that satisfies these trust assumptions is deemed trustworthy. If a user notices that one of her trusted services misbehaves, she will no longer trust it.

By having a set of trusted services, we can prevent location information from flowing to untrusted services. We establish the following rules: For a user query, a trusted service does not forward location information to another service if the located user fails to specify that she trusts it. Similarly, for a room query, a trusted service does not forward location information to another service if the owner of a room fails to specify that she trusts it. Therefore, services are able to get location information only when the located user or the owner of a room approve such a forwarding.

In our design, we make the following assumptions about trust and dealing with it:

- Since having a user specify which of the services she trusts may be cumbersome, especially if there are a lots of services available, we give users the possibility to specify that they trust all the services in a particular organization (e.g., all services run by SCS facilities). This mechanism requires that digital tickets cannot only cover single entities, but also groups of entities. We discuss the implementation of this concept in more detail in Section 6.1.

- All users are supposed to trust the first service to be contacted, which is the People or Room Locator service. If the locating user does not trust it, he should not contact it. If the located user or the owner of a room do not trust it, they can only hope that underlying services refuse to give information to it and thus other users are not able to learn about her whereabouts/occupants.

- As described in Section 3, our location system may fail because of, for example, users leaving their active badge in the office. However, if a user trusts a service, we believe that it is reasonable to assume that she is willing to cooperate with this service. For example, if a user trusts the active badge service, she implicitly promises not to leave her badge in her office or to give it to other people. In short, by allowing users to express which services they trust, we hope that they do not try to fool their trusted services.

5.2 Confidence values

Location information is associated with a confidence value indicating the probability that the information is correct. Typically, the confidence value is set by the same service that generates the location information. For example, if the Login service notices that the user logged in at the console of a computer has been idle for a while, this location information is given low confidence. Similarly, a Wavelan-based service may not be able to determine exactly on which floor or in which room a device is, thus it assigns the returned location information low confidence.

It is possible that other services than the one generating the location information assign it low confidence. The fact that a locating or located user does not trust a service does not imply that the user does not want this particular service to be contacted for answering queries. For example, even though you may not trust the person running the Login service, you do not expect it to actively falsify location information. Therefore, you still want this service to be contacted, but you do not want to have this information high confidence. A service getting location information from some other service assigns it low confidence if neither the locating nor the located user specify that they trust this service (either directly or by trusting the entire organization this service is part of). Since at least the People and Room Locator services, which are the first services in the calling chain, are assumed to be trusted by the locating user, we are guaranteed that there is always at least one service in the chain that adjusts confidence values appropriately.

5.3 Delegation

Entities in our system grant other entities in the system particular kinds of rights. For example, a located server may give a locating user access to her location information. It is up to the located user to also give the locating user the right to forward this right to a third user. In such a case, the located user effectively gives the right to decide about her location policy to the locating user. In the remainder of this paper, we are going to use the term "delegation" when an entity grants access rights to a second entity and it also permits the second entity to grant these rights to a third entity.

We now describe how we exploit delegation in our system in more detail.

5.3.1 Policy specification

The People Locator service delegates the right to establish the location policy of a particular located user to this user. (A similar model is used for room location policies.) As noted above, for a located user, this right is identical to having access to her location information and to be able to re-delegate this access right. A located user can then issue further digital tickets that give other users also access to her location information. It is is up to the located user to decide to whom and at what granularity she gives this access right. In addition, she can also permit the recipients to re-forward the access right to other people. The People Locator service itself does not have to be aware of the identity of the users she gives access rights to, any locating user that can prove that the located user gave him access rights (by presenting his digital ticket) will be granted access. This solution thus makes dealing with unknown users easy. In addition, as mentioned in Section 5.1, digital tickets allow giving access rights to entire groups of people. Therefore, giving a group access to location information is straightforward in this approach.

5.3.2 Policy check

As discussed in Section 2, we assume that each located user defines a user location policy stating which users are allowed to get what kind of location information about her. Each trusted location service is supposed to implement this policy faithfully. However, to reduce overhead, not every service is required to actually run the policy check for a request, a service can delegate this task to some other service which precedes it in the calling chain. If a user trusts a service, she also trusts it to delegate this policy checking to a trustworthy entity. Typically, services within an organization delegate policy checking to the first service in the calling chain (the People Locator service). However, it is up to the People Locator service to also delegate policy checking to some other entity. The service running the policy check approves the request and lets the other services in the calling chain know about the approval so that they are willing to answer the request.

In a similar way, policy checks for room queries can be delegated to the Room Locator or some other service.

Of course, we cannot control how services not in our organization implement policy checking. They might do it in one of the following ways:

- They delegate the policy check to a service in the organization.
- They delegate the policy check to the located user, which then delegates it to a service in her organization.
- They implement their own policy checking.

5.3.3 Device-based location services

Trust as discussed in Section 5.1 mainly addresses information exchange between services that are aware of users. These services need the identity of the located user to be able to find out which services she trusts. However, services such as the Wavelan locator service are not aware of users, since they locate devices. Delegation also lets us overcome this problem. We assume that a device delegates all trust decisions to its owner. A device then implicitly trusts all the services its owner trusts. Therefore, device-based location services can also use trust to decide whether location information should be forwarded to some other service.

5.3.4 Organizations

As already mentioned in Section 5.1, users can choose to trust an entire organization. By doing so, they effectively delegate the decision which services they trust to the organization and they rely on the organization to do the right thing.

5.3.5 Contacted Services

The People and Device Locator service delegate the right to decide about which services to contact for a query to individual users. We let both the located user/owner of a room and locating users specify which location services should be contacted when processing a query. This specification can be per locating and located user, respectively. Alternatively, it can be per user groups. Both the People and Device Locator service are given the list of services to be contacted. Trusted services are expected to implement this list correctly. For a service to be contacted, we require that both the located and locating user include this service in their list. Again, we allow a user to specify that all the services in her organization are contacted. We expect this to be the normal case, assuming all the services are free of charge.

5.4 Discussion

Low-level services such as a Wavelan base station or actual devices that have limited capabilities in terms of CPU and memory resources or whose programmability is restricted may not be able to handle trust decisions at all. Therefore, we tie low-level services to their corresponding high-level services. For example, we associate a Wavelan base station with the Wavelan locator service in the same organization. A Wavelan base station is expected to give location information (and only location information) to only this Wavelan locator service. If a user does not trust a base station, she will not trust the Wavelan locator, either. A similar model can be applied to devices, however, since devices may belong to users, they may not be controlled by the same organization and can impose any kind of restrictions on information flow.

Our proposed solution addresses all the shortcomings of the strawman design listed in Section 4:

Bottleneck. There is no centralized node that has to approve each request. All the services perform access control independently of each other and approve a request only if it is supported by a or multiple digital tickets.

Misbehaving services. A misbehaving service is not given a trust ticket by the located user, thus trusted services further down in the calling chain will not give it any location information.

Unknown users. The location system does not care about the identity of locating users; all it requires is a digital ticket giving access to the locating user. Therefore, locating users can be unknown to the system.

Group access. Group access requires that digital tickets can be given to a group of people instead of only a single person. We provide this feature in our implementation of the digital tickets, as discussed in Section 6.1.

The key question to be answered is whether the strawman design presented in Section 4 can be extended by the concepts of trust and delegation or whether we need a completely different solution. Both concepts require additional data structures to be stored at the centralized server, since neither user trust in services nor delegation of rights can be directly stored in a located/locating user policy matrix.

Even though we could extend the centralized server to make it support trust decisions, the design would still suffer from the other problems mentioned in Section 4, that is, having a bottleneck and unsatisfactory support for groups and unknown users.

Therefore, giving up the approach based on a centralized policy matrix and additional data structures in favor of a distributed solution that requires only one kind of data structures looks more promising. Our data structure is based on digital certificates. We now describe it in more detail.

6 Digital Certificates

To implement the concepts of trust and delegation, as introduced in Section 5, we resort to digital certificates and chains of certificates. In short, users create certificates for each service/organization they trust, services create certificates that delegate location policy checks to the People Locator service, and the People Locator service creates certificates signing away the right to give access to location information about a user to that particular user.

In the remainder of this section, we introduce the concept of SPKI/SDSI certificates, present in an example scenario how they are used in our system, and list some example certificates.

6.1 SPKI/SDSI Certificates

Traditionally, a certificate has been defined as a digitally signed data record containing a name and a public key [8]. The digital signature allows a certificate to be passed around and renders central directories unnecessary. Systems that require access control can exploit certificates for this purpose: upon receiving a request, the signature appended to the request is used for retrieving a certificate from some trusted entity that binds the signer to a name. Then, the name is compared to the names on the local access control list (ACL) and an access control decision is made.

This scheme suffers from two weaknesses: 1) Two steps are required for the access control decision (retrieving the certificate and checking the ACL) and the entity issuing the certificate has to be trusted; 2) It requires globally unique names, that is, a global naming scheme. Such architectures have been proposed (e.g., ISO's X.509), but are from being implemented in a way that they would be useful for everyday use.

Due to these weaknesses, SPKI/SDSI certificates [5] have been recently proposed. They redefine the notion of a digital certificate. SPKI/SDSI certificates do not rely on global names and public keys for authentication, instead, public keys are directly used for access control. An ACL gives access rights directly to a public key. Any request signed with the corresponding private key will get the kind of access specified in the ACL. Therefore, the access control decision requires only one step. In addition, since public keys are globally unique because of their length, we get global uniqueness for free.

SPKI/SDSI also supports delegation of rights. The owner of the public key who is granted access can create a certificate giving all or a subset of her rights to some other public key (i.e., his owner). This process can be repeated multiple times, thus, what we end up with is a chain of certificates anchored at the local ACL. To make the access control decision for a request, a service has to go through the chain of certificates, starting at its ACL and ending at a certificate authorizing the signer of the request. For each certificate, it needs to check whether it delegates enough access rights to its successor so that the request can be granted access.

In addition, SPKI/SDSI supports restriction of delegation. An entity cannot delegate its access rights to some other entity if the original issuer of the access rights does not agree with this delegation. However, note that even though delegation may be prevented within the system by SPKI/SDSI, this restriction does not prevent an entity from forwarding information it is able to retrieve with the help of its access rights to some other entity through channels the system is not aware of (as outlined in the example in Section 2).

SPKI/SDSI typically gives authorizations to public keys. However, it also supports the concept of name certificates that allow a local name (i.e., the name does not have to be globally unique) to be bound to a public key. This binding lets the local name become globally unique and thus it can be directly used as a target for authorization in a certificate. This naming concept is important for cases

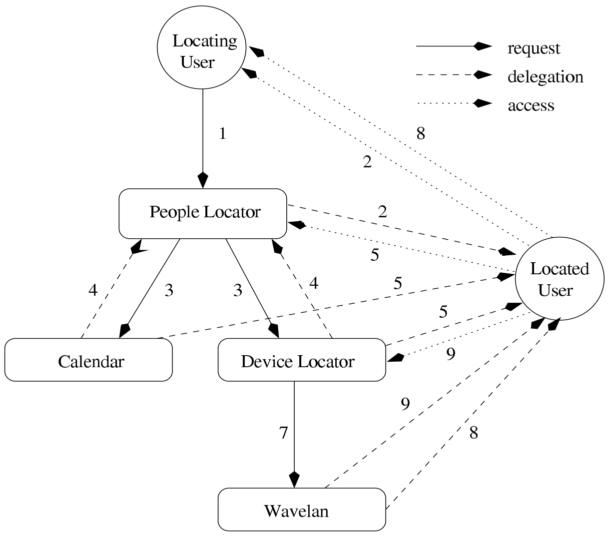


Figure 2: Processing of a request.

where authorization is given to entire groups. We present an example that demonstrates the usage of these name certificates for authorization of groups in Section 6.3.2.

6.2 Example Scenario

It takes several steps to process a request for location information. Each step requires different ACLs and certificates to let handling of the request proceed to the next step. In this section, we present the processing of a request by our location system in a walk through and list the needed ACLs and certificates. In our example, a locating user inquires about the location of a user and the request is processed by the system shown in Figure 2. The location information flows in the reverse direction of a request (not shown in the figure).

As outlined in Section 5.3, we differentiate between simply giving someone access to a resource and also allowing the recipient of the access right to re-forward this access right. In the figure, we denote the first type of giving access with "access" arrows and the second type with "delegation" arrows.

In the initial step (not shown), entities in the system give access rights to other entities and they delegate policy decisions, using either ACLs or certificates:

The administrator of the People Locator service sets up a "location policy ACL" that delegates to individual users the right to decide about their location policy. The administrators of the calendar service and the Device Locator service each generate a "policy checking ACL" that delegates the location policy check to the People Locator service. They also establish each a "trust ACL" saying that it is up to a located user to specify which services she trusts.

The Wavelan service is not administrated by the same entity as the People Locator service and the calendar service. Therefore, the administrator of the Wavelan service decides to have it implement its own location policy checking, thus he does not set up a policy checking ACL, but a location policy ACL. In a trust ACL, he delegates the trust decision for a device to the public key of the device. The owner of the device uses the device's private key to re-delegate these rights to himself in a signed certificate.

A located user creates various certificates to express her location policy. As mentioned in Section 5.3, she can either give only the rights to her location information to some other user or she can also delegate the right to re-forward these rights to another user and thus have this user decide about her location policy. The located user also certifies that she trusts the various location services. By doing so, she effectively gives these services the right to get her location information

In this work, we do not address the question where the certificates generated above are stored in our system. A possibility is to store them in a repository from which services attempt to retrieve the certificates required for granting access to a request. Another possibility is to require a client to keep certificates issued to her with her. When issuing a request, he needs to present all the required certificates together with the request. It is important to note that since certificates are signed, there is no need to keep at them at a centralized or trusted service.

Given this setup, a request is processed as follows:

1) The locating user sends a signed request to the People Locator service inquiring about the location of a particular located user.

2) The People Locator service checks the located user's location policy by building a delegation chain. The first element in the chain is the location policy ACL giving the located user access to her location information, the second one a certificate issued by the located user to the locating user. In general, all the entries in the delegation chain need to give the right to the located user's location information to the next entry entry in the chain and each entry (with the exception of the last one) also needs to give the next entry the right to re-delegate this right.

3) If the delegation chain can be established, the request is forwarded to the calendar service and the Device Locator service.

4) The calendar service and the Device Locator service each build a delegation chain from their policy checking ACLs to the People Locator service to verify whether the People Locator has been given the task to perform the policy checking.

5) The calendar service and the Device Locator service each build a delegation chain from their trust ACLs to the located user and from there to the People Locator service (using the trust certificate issued by the located user) to verify whether the People Locator service is trusted by the located user and thus should be given access to the located user's location information.

6) The calendar service processes the request (not shown in the figure).

7) The Device Locator service determines the wireless devices the located user is currently carrying with her. For each device, a request is sent to to the Wavelan service.

8) The Wavelan Locator does its own policy checking. Similar to 2), it builds a delegation from its location policy ACL to the locating user.

9) The Wavelan service builds a delegation chain from its trust ACL via the located user to the Device Locator to verify whether the Device Locator is trusted.

10) The Wavelan Locator processes the request (not shown).

6.3 Example Certificates

In this section, we present actual certificates and ACLs used for implementing the trust and delegation mechanisms outlined in Section 5. We need a language for defining ACLs giving rights to public keys and for certificates re-delegating these rights. In addition, we also require a tool that, based on these ACLs/certificates, decides whether a request for location information should be granted access. SPKI/SDSI defines such a language and also provides tools to run the access control check. We now present a representative sample of the actual certificates in our location system. More specifically, we discuss the usage of SPKI/SDSI certificates for delegation of policy checks, location policy decisions, and trust decisions. Similarly, SPKI/SDSI certificates can also be employed for specification of the set of contacted services.

6.3.1 Delegation of Policy Check

In step 4) of the example scenario discussed in Section 6.2, the Device Locator service checks whether it has delegated the policy check to the People Locator service. This delegation is expressed in the ACL given below. It is kept at the Device Locator service. The ACL states that user location policy checking (which is described after the keyword `tag` and given the name `policy_check user` here) is delegated to the People Locator service, more specifically to the owner of the public key `pub_key:people_locator`. Note that `pub_key:people_locator` would be replaced by the actual public key of the People Locator service in a real implementation. The keyword `propagate` states that the People Locator service is allowed to delegate the given right to some other entity by issuing a certificate.

```
(acl
  (entry
    (pub_key:people_locator)
    (propagate)
    (tag (policy_check user))
  )
)
```

Given the ACL above and a request for location information signed by the People Locator service, the Device Locator service will conclude that the People Locator service has run the location policy check and that the user is allowed to get the requested location information.

6.3.2 Policy Check

In the next examples, we show how the People Locator service lets located users specify their location policy and how policy checking is implemented. In the example scenario, this verification corresponds to step 2). For each located user, the People Locator service keeps an entry in its local ACL that delegates the user's location policy decision to that particular user. In the example below, user Alice's location policy decision is delegated to Alice. Note that Alice is given only the right for her location policy, but not for other people's location policy. This condition is expressed by including `alice` in the `tag` section.

```
(acl
  (entry
    (pub_key:alice)
    (propagate)
    (tag (policy alice))
  )
)
```

In the certificate below, Alice (`issuer`) gives Bob (`subject`) access to her location information. Note that a certificate has to be used for this delegation (since it is given out to Bob) and that the certificate has to be accompanied by Alice's signature (not shown here). Bob can locate Alice only if she is either in Wean Hall or in Room 1234 in Doherty Hall and on Monday and Tuesday between 8am and 12pm and between 1pm and 6pm. In addition, Alice delegates only a subset of her access rights to Bob, that is, he gets only coarse-grained access to her location information.

```
(cert
  (issuer (pub_key:alice))
  (subject (pub_key:bob))
  (propagate)
  (tag
   (policy alice
    (* set (* prefix world.cmu.wean)
           (world.cmu.doherty.room1234))
    (* set (monday
           (* set
           (* range numeric
              (ge 800) (le 1200))
           (* range numeric
              (ge 1300) (le 1800))))
           (tuesday
           (* set
           (* range numeric
              (ge 800) (le 1200))
           (* range numeric
              (ge 1300) (le 1800))))
    )
   coarse-grained))
)
```

Given a location request signed by Bob and the ACL and certificate above and if the location and time constraints are met, the People Locator service will conclude that Bob is allowed to get coarse-grained location information about Alice.

Note that as explained in Section 2, Alice might decide not to give Bob the right to forward his access rights to some other person. She would then omit the `propagate` entry in the certificate.

Room location policies look similar to user location policies. An example `tag` section is given below.

```
(tag
 (room_check wean.8220
  (* set (alice) (bob))
  (* set (monday
         (* range numeric (ge 800)
                          (le 1700)))
  anonymized)))
```

There are three differences: First, the name of the located user is replaced by the name of the queried room. Second, the set of locations is replaced by a set of users. A room query will return only people listed in this set. Third, there are three granularity levels: `unrestricted` specifies that a querying user is allowed to get the actual names of the people in the room, `anonymized` states that a querying user gets only the information that someone is in the queried room, and `policy` indicates that information should be returned only if the located user grants this information flow in her user location policy. As explained in Section 2, we do not expect the owner of a room to have the right to re-delegate his access rights, so the certificate would not include a `propagate` entry.

For user location policies, we currently constrain information access based on the identity of the locating user, the current time, and the current location of the located user. There are additional constraints that might be of interest when formulating a location policy. For example, the location preferences of a located user could depend on the task she is currently executing ("I do not want to be located when I am doing something important.") or the role she currently is in (e.g., police officer on duty vs. police officer off duty). These task/role constraints could be added to the `tag` section, in addition to or instead of the time constraints. A trusted entity would then have to determine the current task/role of a located user so that it can be taken into account for access control.

As mentioned in Section 6.1, SPKI/SDSI supports the concept of name certificates to delegate rights to entire groups of people. We now present an example in which access to location information is given to a group of people without having to list each group member explicitly in the delegation certificate.

Alice decides that all her friends should get coarse-grained access to her location information. She first has to define who her friends are. For each of her friends, she issues a name certificate assigning the name `friend` to this friend. If Bob is her friend, she would release the following certificate:

```
(cert
  (issuer (name pub_key:alice "friend"))
  (subject (pub_key:bob))
)
```

To give all her friends coarse-grained access to her location information, Alice releases the following certificate:

```
(cert
  (issuer (pub_key:alice))
  (subject (name pub_key:alice "friend"))
  (propagate)
  (tag (policy alice ...))
)
```

The subject in the certificate above is no longer a public key, but a name (bound to a public key). Note that the issuer of the certificate delegating access rights and of the name certificate do not have to match. For example, Alice could give access rights to Bob's sister, whereas the name certificate binding the public key of Bob's sister to the name `sister` would be created by Bob.

6.3.3 Trust

In order to actually deliver location information to the People Locator service, a trusted service requires a certificate from the located user or the owner of a room saying that she trusts the service requesting the information (or the entire organization). Next, we give an example for such a certificate. This certificate is used in step 5) of the example scenario. There also needs to be a corresponding ACL stored at the People Locator service (not shown here).

```
(cert
  (issuer (pub_key:alice))
  (subject: (pub_key:people_locator))
  (tag (trust alice))
)
```

7 Implementation Status and Deployment Issues

We have implemented a subset of the location system shown in Figure 1. The system consists of the People Locator service and a location service that proxies to SCS's centralized calendar system to use it as source for location information. We are currently working on integrating a Wavelan-based location service into our system. Authentication of services to clients and confidentiality of information is achieved with SSL-based connections. Access control is entirely based on SPKI/SDSI. Location information and SPKI/SDSI certificates are transmitted between services using the Aura API, which is a protocol running over HTTP and which exchanges messages encoded in XML.

Integrating a service based on a new kind of location technology into our location system requires the following steps: The service has to be extended to support the Aura API and SSL. As an access control mechanism, it needs to be able to validate a SPKI/SDSI certificate chain. The administrator of the system needs to establish local ACLs and, if desired, delegate the policy check to some other service. Finally, the new service needs to be advertised to users so that they can specify whether it should be used for their queries and whether they trust it.

In our current solution, digital certificates have to be created with a command-line tool. For further deployment of our location system, we need to build a graphical user interface that makes certificate generation transparent to the user of the system. In addition, we also require a certificate repository and a mechanism for automated building of the certificate chain needed for an access control check. Finally, we need to investigate how current authentication mechanisms such as Kerberos can be integrated into our system.

8 Related Work

Several location systems, all of them based on only one location technology, implemented only within one administrative entity, and/or not addressing the various security issues mentioned in this paper have been proposed [2, 6, 11, 13]. Notable exceptions are the location systems designed by Spreitzer and Theimer [12] and Leonhardt and Magee [9].

Spreitzer and Theimer's location system [12] is based on multiple technologies and they employ "User Agents" for privacy control. Each user has her personal agent that gathers location information about her and that processes requests for this information. It is up to this agent to implement access control. The system is designed to work in an environment with different administrative entities, although the actual implementation runs only within a single entity and the authors do not mention how users specify services they trust. For room queries, the system allows users to register with "Location Brokers". As an important difference from our system, the amount of information and the users granted access to room queries is determined by the located user, not the "owner" of the room.

Leonhardt and Magee [9] also address security considerations. Based on the observation that user queries can be implemented by a series of room queries and vice versa, the authors argue that access control for both types of queries needs to be consistent. The authors propose an extension to the matrix-based access control scheme to implement this requirement. As outlined in Section 2, we believe that having consistent user and location policies is difficult to achieve when these policies are established independently of each other. Having them establish in a synchronized way is difficult since a located user and the owner of a room may have completely different views about the kind of access control required. Unfortunately, Leonhardt and Magee do not discuss how policies are established in their system.

SPKI/SDSI certificates have already been employed in some other systems for access control. For example, Maywah [10] describes how SPKI/SDSI is used for access control in a Web client. Howell and Kotz [7] present three example applications that rely on SPKI/SDSI certificates; a Web server, a database, and a proxy.

Similar to SPKI/SDSI, KeyNote [4] and its predecessor PolicyMaker [3] also provide a language for specifying access control policies and a tool for checking whether a request should be granted access. As in the case of SPKI/SDSI, public keys are authorized directly. KeyNote does not support name certificates; it is up to the application to verify the binding between a name and a public key (i.e., the public key corresponding to the private key used for signing a request). In addition, KeyNote does not allow restriction of delegation.

9 Conclusions and Future Work

In this paper, we have analyzed the security requirements of a secure location system and presented the design of such a system. Our solution relies on three key concepts: trust for dealing with misbehaving services, confidence values for dealing with inaccurate information and information retrieved from untrusted services, and delegation for delegating policy decisions and policy check decisions to other entities in the system.

We have been able to formulate all of our policy, trust, and delegation decisions using a single data structure: SPKI/SDSI certificates. Therefore, these certificates provide a high degree of flexibility. A ubiquitous computing environment poses new challenges on access control that cannot be easily satisfied by conventional mechanisms. We believe that due to their flexibility, SPKI/SDSI certificates are a promising approach and deserve further investigation on their usability in such environments.

In future work, we want to offer access to our location system to a bigger community of users, so that we can incorporate their feedback on usability into our system.

References

[1] http://www.cs.cmu.edu/~aura/.

[2] P. Bahl and V. Padmanabhan. RADAR: An In-Building RF-Based User Location and Tracking System. In *Proceedings of Infocom 2000*, pages 775–784, March 2000.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of 17th IEEE Symp. on Security and Privacy*, pages 164–173, 1996.

[4] M. Blaze, J. Ioannidis, and A. Keromytis. The KeyNote Trust-Management System Version 2. RFC 2704, September 1999.

[5] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. RFC 2693, September 1999.

[6] A. Harter and A. Hopper. A Distributed Location System for the Active Office. *IEEE Network*, 8(1), January 1994.

[7] J. Howell and D. Kotz. End-to-end authorization. In *Proceedings of OSDI 2000*, pages 151–164, October 2000.

[8] L. M. Kohnfelder. Towards a Practical Public-key Cryptosystem. MIT S.B. Thesis, May 1978.

[9] U. Leonhardt and J. Magee. Security Considerations for a Distributed Location Service. *Journal of Network and Systems Management*, 6(1):51–70, March 1998.

[10] A. Maywah. An Implementation of a Secure Web Client Using SPKI/SDSI Certificates. Master's thesis, MIT, 2000.

[11] N.B. Priyantha, A. Chakraborty, and H. Balakrishnan. The Cricket Location-Support System. In *Proceedings of Mobicom 2000*, August 2000.

[12] M. Spreitzer and M. Theimer. Providing Location Information in a Ubiquitous Computing Environment. In *Proceedings of SIGOPS '93*, pages 270–283, Dec 1993.

[13] A. Ward, A. Jones, and A. Hopper. A New Location Technique for the Active Office. *IEEE Personal Communications*, 4(5):42–47, October 1997.

Network Aware Data Transmission with Compression

Ningning Hu *
Computer Science Department
Carnegie Mellon University
(hnn@cs.cmu.edu)

Abstract *Network aware application* can achieve better performance by dynamically adapting to network service changes. The key question for network aware application development is how to obtain information about the performance of different system module. In this paper, we consider an important category of network aware application – *Compressed Data Transmission*. Compression can reduce network transmission time by reducing the size of data to be transmitted, but on the other hand it increases local processing overhead. The tradeoff between increased network processing and decreased local processing is critical to application's decision on how to transfer data. In this paper, we present our model to make such decision and discuss the methods of detecting network resources and predicting compression performance parameters. Experimental data on local testbed is presented to evaluate our methodology. We also discuss an improved model on how to deal with the overlap between network transmission and local processing, which has the potential to improve the application performance.

1 Introduction

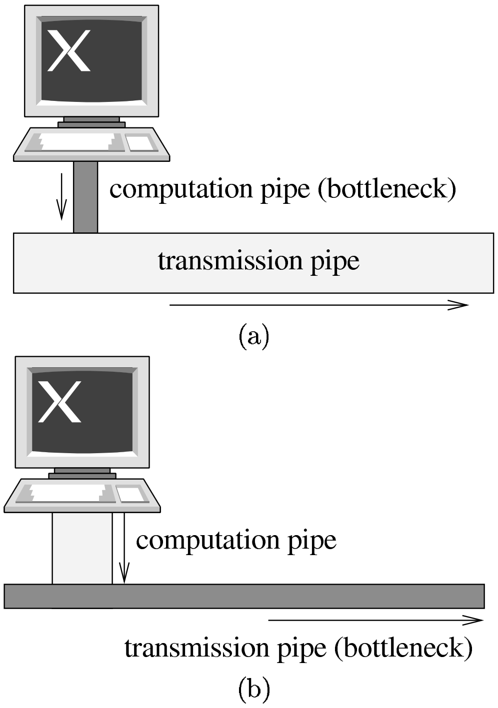
The Internet is well known for its unpredictable performance, people expect to experience different level of Internet service in different places and at different times. When users can choose from a number of sites providing the same service, which is often the case on current Internet, they generally have no idea which one is the best to choose. *Network Awareness* is one of the techniques to deal with these problems. It enables an application to change its behavior according to network performance, allowing it to achieve consistent performance over a diverse sets of networks and under a wide range of network conditions.

In this paper, we focus on an important category of network aware application – *Compressed Data Transmission*. This type of application has the ability to compress data before sending it out with the potential to reduce network transmission time and reduce the total application elapsed time. An example scenario in which *Compressed Data Transmission* can be useful is as follows. Suppose we are leaving for a conference by air. Just before boarding, we want to transfer a big file back to our office machine through a wireless LAN. But at the same time, another plane arrives, a lot of people get off. They start using all kinds of mobile communication devices to send and receive voice and digital messages, which saturates our wireless transmission channel. Under this condition, compression may be an important tool to help us in finishing our work without missing the flight.

Compression can reduce the transmission time by reducing the amount of data to be transferred. But it also increases the local processing time by introducing the compression overhead. Whether or not an application can benefit from data compression depends on the tradeoff between the reduction of network transmission time and the increase of local processing. In another way, we can think the whole procedure of data transmission as composed by two data flow pipes (Fig. 1). The first one is the data flow from local host to network interface. The rate of this flow is determined by the host processing capability. The other pipe is the data flow on the network. The rate of this pipe, that is, the data transmission rate on the network, is determined by the available bandwidth. The performance of the two pipes together determines the performance of the application. For example, in Fig. 1(a), the available bandwidth is large enough, and the network pipe could transmit data faster than the host pipe. Under this condition, the performance of the application is largely determined by the host pipe rate. Similarly, in Fig. 1(b), the application performance is determined by network pipe rate. Determining which one is the bottleneck during the execution, need some techniques to calculate the concrete network transmission time and local computation time.

In this paper, we describe our methods to detect and predict network performance and compression

*Ningning Hu is advised by Prof. Peter Steenkiste.



Data flow pipes involved in *Compressed Data Transmission*. In (a) the bottleneck is on the local computation pipe, while in (b) the bottleneck is on the network.

Figure 1: Data flow pipes

overhead. A simple model using these techniques to prediction the application performance is presented. Experiment is carried out to evaluate the performance of the simple model. We will see that, although in most cases the simple model could make the right judgment for application, the difference between predicted values and measured values is significant sometimes. Analysis of the prediction error tells us that the overlap between different types of processing modules must be considered to make a correct judgment. Based on these considerations, we propose an improved model, which explicitly considers the effect of different performance bottleneck.

This paper is organized as follows. Section 2 abstracts *Compressed Data Transmission* into a simple application prototype, and discusses the detailed techniques to predict compression performance and network performance. In Section 3, experimental data and analysis are presented to show the application performance. Section 4 is an extended discussion of the experimental data, and the modeling of the overlap between network data transmission and local processing, which can improve the prediction performance. Section 5, 6 and 7 talks about the related work, conclusion and future works, respectively.

2 Architecture

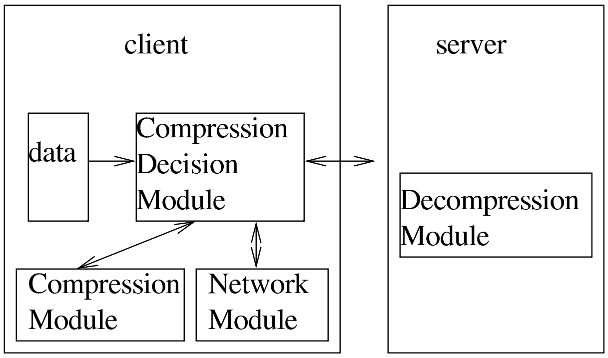


Figure 2: Application architecture

The application used in our study is a simple data transmission application (Fig. 2). It has two functions: compression and transmission, with the cooperation of the following three modules:

1. *Compression Module*: provides the functionalities for data compression, and is also responsible for providing compression algorithm related parameters, i.e., *compression speed* and *compression ratio*.

2. *Network Module*: monitors the network performance properties and provides necessary information for the application, such as *available bandwidth*.

3. *Compression Decision Module*: makes the final decision whether the application can benefit from data compression.

To send out a large amount of data, the application first splits it into multiple chunks, and processes each chunk separately. For each chunk of data, it first calculates the total execution time with and without compression using the parameters from *Network Module* and *Compression Module*, compares them and makes the decision how to transfer the data. *Network Module* provides the *available bandwidth* information, which is used to compute the network transmission time; *Compression Module* maintains the empirical data related with the specific compression algorithm – *compression speed* and *compression ratio*, which are used to calculate the local compression time.

If the application decides not to use compression, the original data will be simply sent out. Otherwise, it will be "forwarded" to the *Compression Module*, where it is compressed, before being sent to the server. During that procedure, a simple application

```
repeat {
        COMPRESSED_DATA_TRANSMISSION(data);
  }     until (all data is sent out);

  COMPRESSED_DATA_TRANSMISSION(data)
  {
        compr_speed = GET_SPEED();
        compr_ratio = GET_RATIO();
        cur_bw = GET_CUR_BW();
        if (COMPRESSED_TRANSFER(data, cur_bw,
              compr_speed, compr_ratio)){
              COMPRESS(data, &compr_data);
              SEND(compr_data);
        }
        else {
              SEND(data);
        }
  }
```

Figure 3: Application pseudocode

protocol is used to tell the server the transmitted data needs decompression. The pseudocode for the application is shown in Fig. 3.

2.1 Compression Decision Module

Different applications may use different ways to make decision about whether or not to use compression. Some applications may choose to use compression as long as the total execution time for the data transfer with compression is smaller than transmission time without compression, with the objective of reducing the load on the network. Other applications focus on achieving best data quality, so they try to avoid using compression as long as the data transmission without compression is below some threshold, since some compression algorithm may lose information,

No matter what policy the application uses, it must calculate the total execution time for both the compressed mode and the uncompressed mode. In this paper, for the uncompressed mode, the total execution time is simply the data transmission time, which can be computed as:

$$comm\_time = \frac{data\_size}{available\_bandwidth} \tag{1}$$

where *data_size* is the size of the data to be transmitted by the application, and *available_bandwidth* is the available bandwidth of the network link.

The overhead involved in the data transmission with compression is computed as:

$$total\_time = compr\_time + send\_time \tag{2}$$

$$compr\_time = \frac{data\_size}{estimated\_compr\_speed} \tag{3}$$

$$send\_time = \frac{compr\_size}{available\_bandwidth} \tag{4}$$

$$compr\_size = \frac{data\_size}{estimated\_compr\_ratio} \tag{5}$$

where *estimated_compr_speed* and *estimated_compr_ratio* are the two parameters provided by *Compression Module.*

The above methods to predict the application performance shown in formula (1) - (5) is denoted as *simple model* in the following sections.

2.2 Compression Module

Our prototype uses the general purpose lossless compression algorithm - *gzip*[7] in the *Compression Module. Gzip* is a standard compression utility commonly used on Unix operating system platforms. It does not consider domain specific information and uses a simple, bitwise algorithm to compress file. It can work in stream mode, that is, at the time that the compression starts, not all data has to be available. In our implementation, we simply incorporate the *gzip* library into our application code. The performance parameters that this module needs to provide are *compression speed* and *compression ratio.* We use empirical method to get their value; see Section 3.2 for details.

2.3 Network Module

We use Remos in the *Network Module* to monitor and predict network performance. In the following, "network performance" means *network bandwidth available for the application.*

Remos (Resource Monitoring System)[3, 4, 15] is a network performance middleware service developed at CMU. It provides a scalable, flexible and portable network monitoring system for applications in distributed computing environments. Remos is composed of two parts: *Modeler* and *Collector.* The *Modeler* implements the Remos API, which enables applications to communicate with the *Collector*, query the interested information, and transform the data from the *Collector.* The *Modeler* also integrates some prediction services [4], allowing history-based data collected across the network to be used to generate the predictions needed by a particular user. The *Collector* is responsible for the network performance information collection, using SNMP[18, 19] or benchmarks. The network performance information includes network topology, link latency, link capacity and link available bandwidth. Several types of collectors are implemented in Remos. They are the *SNMP Collector*, *WAN Collector*, *Bridge Collector* and *Master Collector* [15]. Different collectors work in different network environment and provide different monitoring methods.

Our experiments are carried out on a local LAN testbed. We use the *SNMP Collector*[15] to get the available bandwidth information. In a LAN, the *SNMP Collector* depends on SNMP agents[19] to collect information. SNMP agents can provide the information about the total amount of input data and output data from each network interface. By keeping track of these values, the *SNMP Collector* can estimate the average throughput of the link between two end hosts during the measurement period. Together with the knowledge of the link capacity, which can also be obtained from SNMP agents, it will be able to figure out the available bandwidth. Fig. 4 illustrates this method.

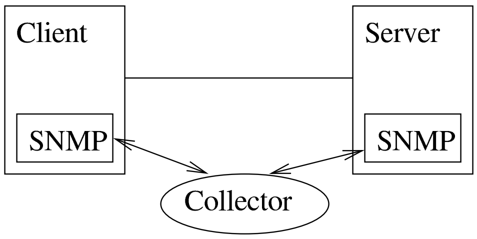


Figure 4: Available bandwidth prediction by Remos SNMP Collector

To predict the available bandwidth for the near future, the *SNMP Collector* periodically queries SNMP agents and keeps a record history (associated with a *history period* time slot). It uses the average of the history record in the history period as the prediction for available bandwidth.

3 Experiment on the Simple Model

In this section, we discuss our evaluation of the performance of the simple model. We first give give some data about the performance of compression module and network module, then discuss the experiments for data transmission with and without compression. We will see that the simple model could give very accurate judgment on whether or not to use compression. But we also notice the significant difference between the predicted values and the measured values, which we think is due to the defect of the simple model. In Section 4, we explain the causes of the prediction error and propose an improved model for making predictions.

3.1 Experimental Setup

Experiments are carried out on our local lab testbed. Fig. 5 shows our experiment configuration. There are four host machines, the two slashed squares are the application client and the application server, and the two grid squares are the competing client and the competing server. The two circles are routers, and the link between the routers is the bottleneck link in this simple network; its nominal transmission capacity is 10Mbps. The application client and the application server are executed on two Digital Unix machines. The competing client and the competing server are used to create the competing flow so as to change the available bandwidth on the link. The application uses a single TCP connection to transfer data. While the traffic generated by the competing hosts (also called *competing flow*) uses UDP packets since it is more accurate to predict the available bandwidth when the background traffic is a UDP flow. A *SNMP Collector* is deployed to monitor the available bandwidth.

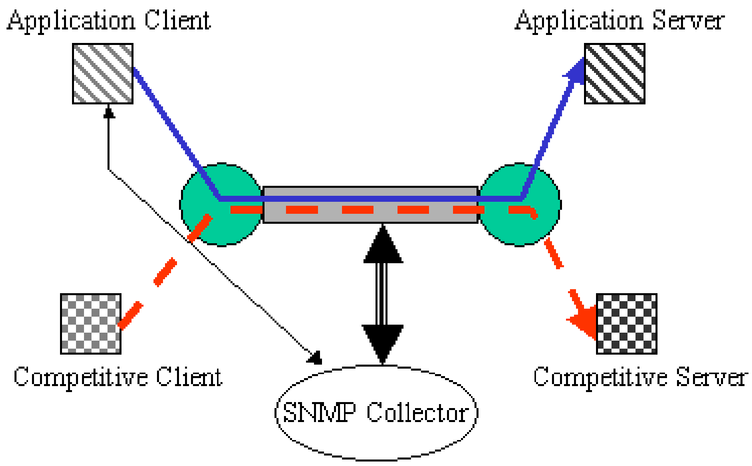
In the experiment, we focus on the comparisons of four pairs of parameters:

1. *predicted_total_time* and *total_time*
2. *predicted_compr_time* and *compr_time*
3. *predicted_send_time* and *send_time*
4. *predicted_compr_size* and *compr_size*

Here, *predicted_total_time*, *predicted_compr_time*, *predicted_send_time* and *predicted_compr_size* are the values computed according to formulas (2) - (5). *Total_time*, *compr_time*, *send_time* and *compr_size* are the corresponding measured values in our experiment.

3.2 Compression Speed and Compression Ratio

Compression speed is related to the data format and the machine type. The relationship between application performance and host machine parameters is a research topic that is outside of the scope of this paper. During the experiments, we keep using the same



The four squares are host machines, the slashed squares denote the application client and the application server, and the two grid squares are the competing client and the competing server. The two circles are routers.

Figure 5: Experiment setting

	Compression Speed		Compression Ratio	
	Average (MBps)	Std. Dev.(MBps)	Average	Std. Dev.
TXT	0.84569	0.10470	4.0676	1.2047
PS	0.77839	0.18261	3.8816	2.5839
Binaray	0.65497	0.04605	2.0746	0.2169
PDF	0.87562	0.07125	1.1856	0.03035
JPG	0.83335	0.01509	1.0075	0.0099

Table 1: Compression speed and compression ratio of *gzip* with 16KB compression buffer

machine for all the compressions, and make sure that our application is the only workload. This way, we can think of *compression speed* as a function of compression algorithm. The *compression speed* is also affected by compression buffer size, but we omit this factor by using the same size of buffer, which is 16KB.

The method to get the *compression speed* and *compression ratio* is as follows. Take the same type of data, compress them using *gzip* and record the measured value. By *data type*, we mean the type of data file, for example, binary code file, postscript file, text file, JPEG file, etc. Table 1 presents the empirical data that we get using this method. For each type of data, we randomly download 100 - 110 files from the Internet, and for each data file, repeat the same compression procedure six times. From Table 1, we can see the standard deviation is quite small, which make us believe file type is a reasonable way to differentiate data when considering the general purpose compression algorithm – *gzip*. In our experiment, the source data is a big tar file of a collection of binary files, and we simply use the value from Table 1, indexed by data type (file type).

3.3 Available Bandwidth

In the experiment, the application depends on Remos to provide available bandwidth information. Clearly, the accuracy of the network performance information can affect the final prediction accuracy significantly. We first use a simple experiment to get an idea of the accuracy of the Remos information.

Fig. 6 shows the experimental results. In the experiment, we keep monitoring the available bandwidth of the network path between two machines on our testbed, which generally carries no traffic. The measured value is plotted using the solid line in the figure. At around 6 second (*competing flow starts* in Fig. 6), a 6Mbps UDP flow is added to generate the competing traffic. We can see, with the introduction of competing traffic, Remos starts reporting reduced available bandwidth. After around 4 seconds, the reported value gets stable, with the final value around 3.4Mbps. We think it is reasonable since the link capacity is set as 10Mbps.

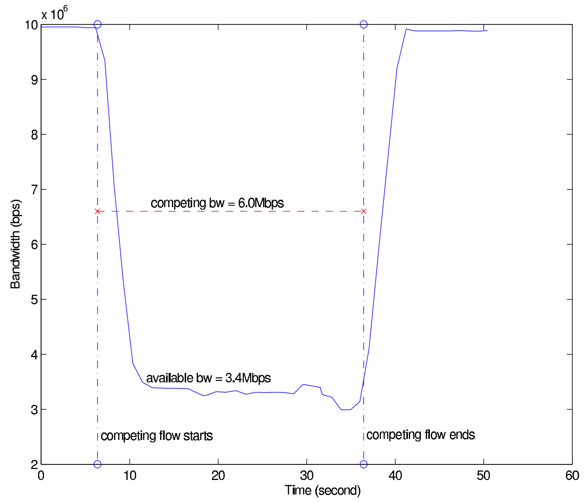


Figure 6: Accuracy of Remos available bandwidth prediction

The reason that we use UDP traffic instead of TCP traffic as the competing flow is that UDP traffic sender side throughput is not affected by other traffic on the same path. In our experiment, we need different rates of competing traffic, so we can easily control its traffic throughput.

3.4 Application Performance

To evaluate application performance, we transfer the same amount of data with compression and without compression, recording the execution times.

repeat {
  COMPRESS(*data*, &*compr_data*);
  SEND(*compr_data*);
} **until** (all data is sent out);

Figure 7: Pseudocode for data transmission with compression

repeat {
  SEND(*data*);
} **until** (all data is sent out);

Figure 8: Pseudocode for data transmission without compression

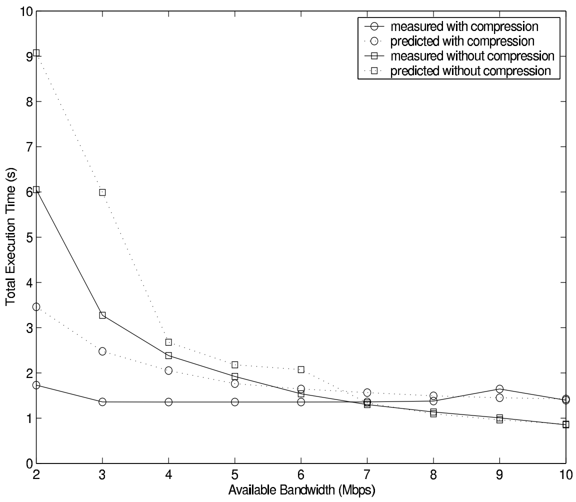
In order to know whether the application would make the right decision for using compression, we did experiments on data transmission with and without compression separately, measuring and comparing their execution time. The experiment pseudocode is shown in Fig. 7 and Fig. 8; the data is split into multiple chunks, and it process one chunk each loop. We believe this is the most general way of transferring large data set. We repeat the same experiment 20 times with the size of the data involved in network transmission increased by 1MB each loop, where the data size starts from 1MB, and the data source is a large binary code file. We take the averaged value as the final measurement. We repeat the same experiment, with competing traffic changing from 0 to 8Mbps.

Fig. 9(a) shows the predicted and measured total execution time for both in compression and in uncompression mode. Although in some cases, the absolute error in this figure is not minor, the predicted relationship between the elapsed time with and without compression is the same as that of the measured values. When the available bandwidth is less than 7Mbps, compression mode is faster than uncompression mode; and the uncompression mode shows its advantage when the available bandwidth is bigger than 7Mbps. That means that the application in Section 2 would make the correct decision.

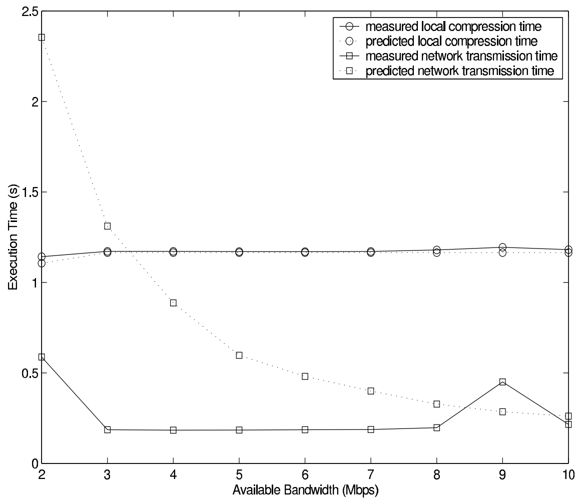
Fig. 9(b) shows the application measured local compression time and the network transmission time when we use compression in data transmission. We can see that the local compression time does not show significant changes, which is easy to understand, since it is not affected by the available bandwidth. But the network transmission time looks strange: it does not change very much as the available bandwidth changes from 3Mbps to 10Mbps, unlike the predicted value, which decreases proportionally with the increase of available bandwidth. The reason, which will be discussed in detail in Section 4, is that with the interleaving of local compression and network transmission, there exists execution overlap between the *Compression Module* and the *Network Module*, and the application measured network overhead is actually not the real data transmission time. That basically invalidates the simple model shown in formulas (1) - (5).

3.5 Breakdown of Compressed Data Transmission Time

In Fig. 9, each data point is the average value of 20 experiment results. In this section, we illustrate the detail of a single experiment by showing the performance of each module of the application, in order to get an idea where the error comes from.



(a)



(b)

(a) gives the change of total execution time for data transmission with (circle points) and without (square points) compression, both measured (solid line) and predicted (dot line) values are plotted; (b) is the breakdown the time of data transmission with compression into local compression (circle points) and network transmission (square points)

Figure 9: Application performance

Fig. 10 shows the experimental results for the compressed data transmission, with 6Mbps of UDP competing traffic. Each point in this figure represents one experiment. The same experiment is repeated 52 times, with the transmitted data size increased by 16KB each time. The four figures show the predicted and measured values for *total execution time*, *compression time*, *sending time*, and *data size after compression*. Circle points are predicted value, and star points are measured value. The prediction for compression time is very accurate (see the second and third figure). It shows big difference only in two cases, which can be explained by temporary host system perturbations. Prediction for data transmission time shows a somewhat large difference with the measured value, we will discuss it in Section 3.6. But the overall error is not significant, as shown in the first figure in Fig. 10. This figure shows that the application prediction error mainly comes from the data transmission part.

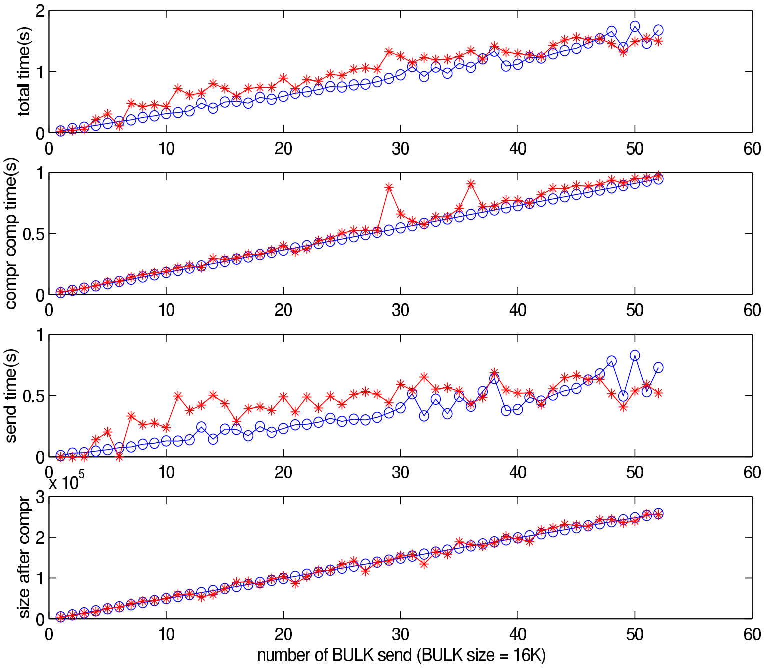
3.6 Analysis

In Fig. 10, we know its error mainly comes from the error of network transmission time. In this section, we systematically evaluate the prediction error of the simple model by making a more complete exploration of the experiment setup. That is, we change the parameters for *Compression Module* and *Network Module*, repeat the same experiment, and measure the prediction errors.

Fig. 11(a) - (d) shows the errors of the prediction model. In Fig. 11(a) and (c), we change the throughput of UDP competing traffics, from 2Mbps to 8Mbps, while keeping the other parameters constant. Fig. 11(a) is the absolute error and Fig.11(c) is the corresponding relative error for *total execution time*, *compression time* and *sending time*.

The prediction error for compression time is less than 4%, and the errors show no relationship with the available bandwidth, which is easy to understand.

Fig. 11(c) shows that the prediction error for the total execution time mainly comes from that of the sending time prediction. Actually, the predicted values of sending time are generally 2-4 times the measured values (this big absolute error only show a small relative error in Fig. 11(c) is because the data sending time is only small part of the total execution time, around 25%). This is because when the socket API sends out data, it first copies the application data into a kernel buffer, and returns after the copying is finished, regardless of whether the transmission has finished. So when we try to measure the processing time of the socket API call, what we get is actually the data copying time, and not the data transmission time. The application could provide data fast enough to make socket API blocks on the socket buffer. When the available bandwidth is high enough (higher than 3Mpbs in Fig.11), the network system can clear the socket buffer fast enough so that the



Circle points are the predicted values, and star points are the measure value. X-axis is the number of data, with transmitted data size increased by 16KB each time. For the Y-axis, the first figure is the total execution time (*total_time*) computed in formula (2); the second figure is the compression computation time (*compr_time*) for formula (3); the third figure is the compressed data transmission time (*send_time*)in formula (4); and the last figure shows the size of the compressed data (*compr_size*) in formula (5).

Figure 10: Breakdown of compressed data transmission execution time

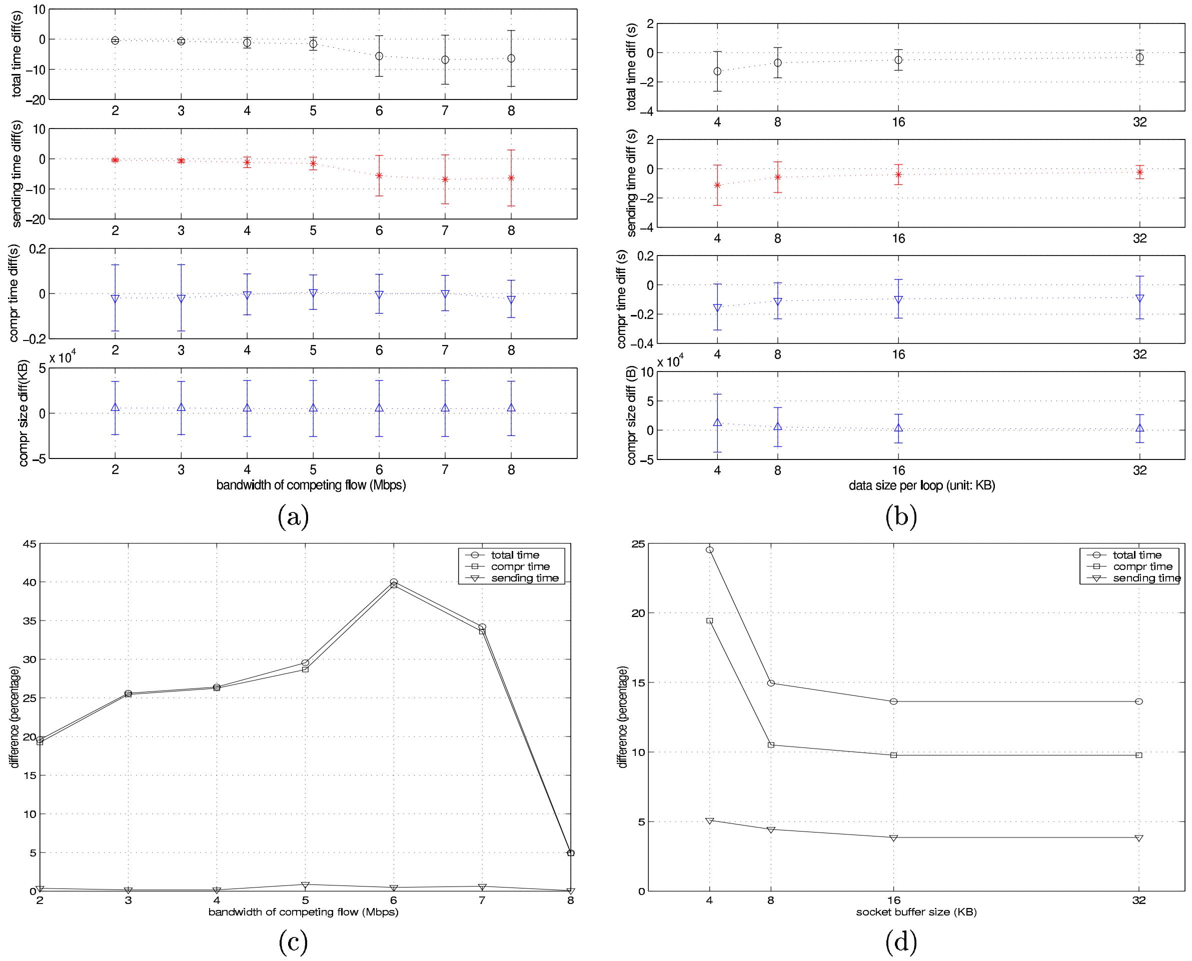
socket API does not block.

Although when the available bandwidth is less then 4Mbps, the socket API blocks due to the slow network transmission rate, the application will start noticing the changes of available bandwidth, the error of link capacity provided by SNMP agent will starts contributing to the prediction error for data transmission. That is, for a link with capacity 10Mbps, the real highest throughput that the application can achieve is actually not exactly 10Mbps, and this error will become more and more significant when reducing the available bandwidth. That is why I see big errors when competing flow bandwidth increases in the second figure of Fig. 11(a).

In Fig. 11(b) and (d), we keep the competing traffic constant, and change the chunk size processed each loop from 4KB to 32KB The four figures show the average difference between predicted values and measured values together with their standard deviation for *total execution time*, *data transfer time*, *compression time* and *compressed data size*. In Fig. 11(d), the prediction error for compression time is also very small, which is less than 10%. And there is large error for data sending time prediction, the reason is similar with that of Fig. 11(c). We also notice the prediction errors tend to reduce with the increasing of data size per loop. It is easy to understand since a larger data size per loop will reduce the number of loops to process data, and fewer loop errors will be accumulated.

4 Model the Overlap of Local Computation and Network Execution

As mentioned in the previous section, in Fig. 9(b), when the available bandwidth is higher than 3Mbps, the execution time with compression does not change very much. This result does not comply with the prediction formulas (2) - (5), which says that with the decreasing of available bandwidth, transmission time should increase, finally increasing the total execution time. In this section, we show that this abnormal results is due to the overlap between network transmission and local compression on the host. More specifically, if the computation time is big enough, what really matters for the total execution time is the socket



In (a) and (c) , X-axis is the competing traffic throughput, and Y-axis is the prediction error of each execution module. In (b) and (d), X-axis is data size processed per phase, and Y-axis is the prediction error for 1MB data processing. (a) and (b) show the absolute errors, (c) and (d) give the relative errors in percentage.

Figure 11: Prediction Error

buffer copying time, not network transmission time. That is why the network available bandwidth will not affect the application performance.

4.1 Theoretical Model

We base our analysis on the execution model in Fig. 7. It is composed of a compression-transmission loop, where each chunk of data is first processed by a computation procedure before it is sent out,

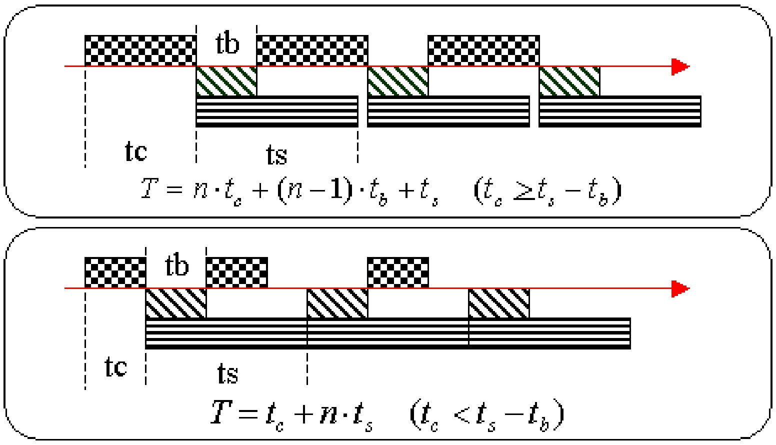
In our implementation, *send()* returns as soon as all the application data has been copied into the kernel buffer. So part of the time used for data transmission will overlap with that of *COMPRESS()*. Assume t_c is the time of compressing one chunk of data, t_b is the socket buffer copying time for the compressed data, and t_s is the corresponding network transmission time. Let us denote the processing time of all the loops as T, then this execution model can be expressed as Fig. 12.

Generally, t_s starts somewhat later than the corresponding t_b, but the time interval is very difficult to measure. So we simply assume that t_s and t_b start at the same time. From Fig. 12, T can be expressed by the following formula:

$$T = \begin{cases} n \cdot t_c + (n-1)t_b + t_s & t_c \geq t_s - t_b \\ t_c + n \cdot t_s & t_c < t_s - t_b \end{cases} \qquad (6)$$

where n is the number of loops in Fig. 7. The relationship between compression time and the total processing time in the above formula is illustrated in Fig. 13

It is easy to see, when computation time is big enough ($t_c \geq t_s - t_b$), t_s is not the dominating factor in the total execution time. Consequently, the



The grid boxes represent local computation, slashed boxes represent local data copying for data transmission, and the line boxes represent the data transmission. The top box shows the overlap when $t_c \geq t_s - t_b$, and the bottom shows the overlap when $t_c < t_s - t_b$.

Figure 12: Overlap between data transmission and local compression

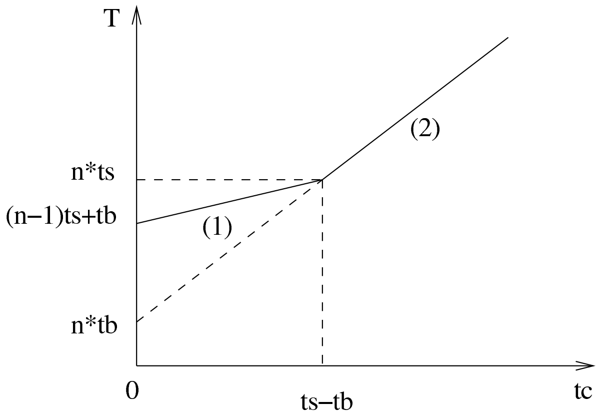


Figure 13: Theoretical model for total execution time, considering the overlap between network processing and local compression

changes of network bandwidth will not affect application performance. The following section shows the experimental result that confirms this claim.

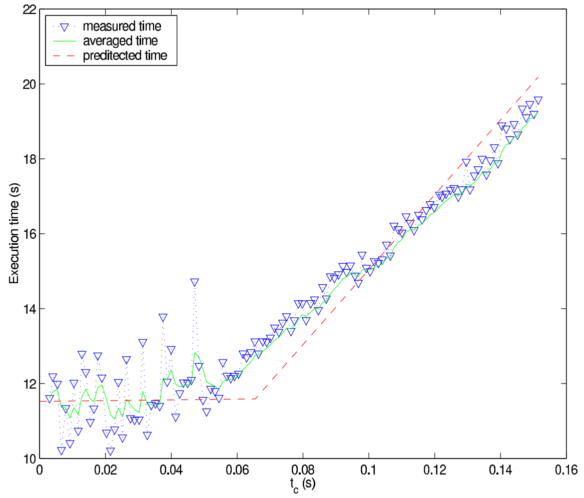
4.2 Experimental Setup

The high level idea of the experiment is the same as that in Fig. 7. We keep running the same compression-transmission loop, measure the execution time of different modules, and try to illustrate the relationship between total execution time and computation time. In our implementation, the computation time should be controllable on a very small time scale, and we should be able to precisely control the computation time, which is very difficult to do for a compression module. With these implementation requirements, we replace the *compression* part in Fig. 7 with a loop of simple arithmetic calculation. The other part of the loop, *send()*, is implemented by a routine socket data sending procedure.

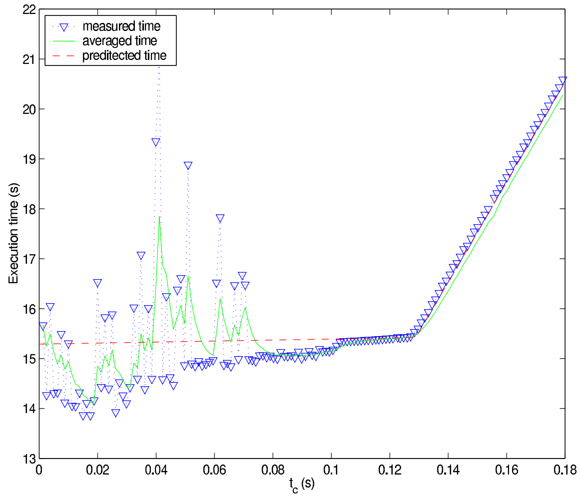
The experimental setting is the same as that in Fig. 5. The bottleneck link bandwidth is 10Mbps, socket buffer size is 16KB, and each chunk of data is also 16KB. A 6Mbps UDP competing flow is added to change the available bandwidth. For each computation time, we repeat the computation-transmission loop 100 times, recording the average value for computation time t_c and buffer copying time t_b.

4.3 Experimental Results

Fig. 14(a) and (b) show the experimental results without and with UDP competing flow, respectively. The dashed line is computed using formula (6). In order to calculate the theoretical value for T, we need to know t_c, t_b, and t_s. The former two can be measured directly in the experiment. t_s is chosen manually from the trace data. We think the measured network transmission time as t_s when t_c is small enough, because the computation part will be completely covered by network transmission. This method of computing t_s is not 100% accurate. The error, we think, leads to the difference between measured value (triangle points) and predicted value (dashed line) in Fig. 14(a). Comparing Fig. 14 with Fig. 13, we can see that the experimental result follows our execution model very well, which we believe confirms the



(a)



(b)

For both figures, X-axis is the computation time t_c (the corresponding t_s is the network transmission time for 16KB data), Y-axis is the total execution time for 100 repetition of the computation-transmission loop. Triangle points are the measured value, and dashed line is drawn according to formula (6). (a) shows the data measured without competing traffic, and (b) shows the data measured with 6Mbps UDP competing traffic.

Figure 14: Experiment about improved model

competing BW (Mbps)	64K data transfer time (s)	estimated available BW (Mbps)
0	0.115	4.5
6	0.155	3.4

Table 2: Estimate the Available Bandwidth

improved model in Fig. 12.

Fig. 14 also provides a way to estimate the real data transfer time t_s (i.e., the *turning point* between segment (1) and (2) in Fig. 13), which can be further used for available bandwidth computation. For example, fitting Fig. 14 into Fig. 13, we can get the data as in Table 2.

Unfortunately, the estimated values does not make much sense. We think it is due to our assumption that socket *send()* (t_s) starts sending data at the same time as that of buffer copying (t_b), which may not be true in reality. The real data transfer time should be smaller than t_s, but the exact difference is difficult to measure.

4.4 Analysis

With the results from formula (6) we can give an explanation for the data in Fig. 9. When the available bandwidth is over 3Mbps, the compression time is larger than $(t_s - t_b)$. That is why the change of available bandwidth from 4Mbps to 10Mbps does not affect the total execution time, and we see no changes in the total execution time with the increase of competing flow bandwidth.

With the improved model, we can improve our prediction model in the following way. We can still use formula (1) to compute the processing time in uncompression mode. But for the compression mode, we should use formula (6) instead of formula (2). Given the data size and the performance parameters from *Compression Module* and *Network Module*, it is not difficult to calculate t_c and t_s, while t_b can be measured directly in the application, that is, the execution time that the application sees from socket function *send()* is actually the socket buffer copying time.

5 Related Work

Network-aware applications are a hot research area which has been widely discussed. This type of application can be classified by their adaptation behavior. [24] gives a discussion about adaptation models used by network aware application, and [1] presented a framework-based approach to develop network aware applications. Some related work in the mobile computing environment is implemented in Odyssey [5, 16, 20, 21].

Compression before transmission and related preprocessing techniques have been used by many network applications [8, 9, 10, 14, 26]. Among them, [8] and [14] discussed two examples which are very similar with our work. [8] talks about the trade-off between the compression ration and the compression time. It presents a system to automatically and dynamically select the compression format to reduce the *Total Delay* based on the future resource performance. Similar to their work, which uses NWS[25] to detect network performance, we also uses an existing network monitoring system to help predict network performance. But we focus on the judgment whether or not to use compression, not compression format, since compression does not necessarily improve the application's performance.

[14] talks about how to use compression techniques in a transcoding proxy in a mobile network environment. By predicting transcoding delay, transcoding size and network bandwidth, it determines whether to transcode and how much to transcode an image for store-and-forward transcoding and streamed transcoding. The problem they focus on is similar to ours, trying to decide which data to send onto the network, but they uses a different way to monitor and predict the network performance, which is very similar with that of [23].

Neither of these works consider the possibility of overlap among different processing modules. Computing the total execution time as the simple arithmetic summary of each module's execution time, in many cases, can impair application performance.

As a key component of the application, available bandwidth prediction is a hot research topic. IDMaps[6] suggests a scalable Internet-wide architecture, which measures and disseminates distance information on the global Internet. NWS[25] is trying to provide accurate forecasts of dynamically changing performance characteristics for a distributed set of metacomputing resources. SPAND[23] determines network characteristics by making shared, passive measurements from a collection of hosts. NIMI[17] proposes to deal with this problem from the perspective of infrastructure, trying to provide a large-scale, extensible platform for network measurement. Besides these systems, which need complicated configuration, simple tools like *bprobe/cprobe*[2], *nettimer*[12], *pathchar*[11] and *sting*[22] are also available for network performance measurement.

6 Conclusion

To adapt to the dynamic change of network performance, applications can use compression to reduce the network transaction time by reducing the size of the data to be transmitted. But compression also increases the local processing time. The trade off between network transmission time and local processing time should be considered to make the right decision whether or not to compress the data.

In this paper, we split the application into three components: *Compression Module*, *Network Module*, and *Compression Decision Module*. We present our method to get the performance information about compression and network bandwidth. Experiments on a LAN testbed shows that our method can work quite well. We can make accurate judgment with our method.

Our experiment also shows that simple aggregation of processing times is not enough to predict the total execution time accurately. We need to consider the overlap between different types of processing. We present a simple model to explain it. Further experiment confirms our model.

7 Future Work

Future work for this paper includes studying of the application's behavior on WAN, since the experiment in this paper is only carried out on the LAN testbed.

We shall also investigate techniques to get better estimation and prediction of available network bandwidth. People have already started looking at this problem. [2, 12, 13] mention some models to do this work. But they all have some limitations in terms of implementation. Accurate and realistic tools to estimate available bandwidth need more work. The improved model discussed in this paper (formula (6) and Fig. 13) actually provides a way to estimate the available bandwidth, but we need to solve some difficult problems before making this model practical, as discussed in Section 4.2.

Another important future work is to make more complete usage of the compression algorithm. The compression algorithm used in this paper, *gzip*, is actually a sophisticated compression algorithm, which allows application to change its compression ratio dynamically. It will improve the application's adaptation ability to network services changes.

8 Acknowledgements

I would like to thank Prof. Peter Steenkiste for his discussion and encouragement on this work; Nancy Miller and Christopher Lin for their assistance in setting up the experiment; Yang-Hua Chu and the anonymous reviewers for their helpful comments on an earlier draft of this paper.

References

[1] J. Bolliger and T. Gross. *A Framework-Based Approach to the Development of Network-Aware Application*, IEEE Transactions on Software Engineering (Special Issue on Mobility and Network-Aware Computing), May 1998, 24(5), pp. 376-390.

[2] Robert L. Carter and Mark E. Crovella. *Measuring Bottleneck Link Speed in Packet-Switched Networks*, TR-96-006, Boston University Computer Science Department, March 15, 1996.

[3] Tony DeWitt, Thomas Gross, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, Jaspal Subhlok, Dean Sutherland. *ReMoS: A Resource Monitoring System for Network-Aware Applications*, Technical Report, CMU-CS-97-194.

[4] Peter A. Dinda, Thomas Gross, Roger Karrer, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, Dean Sutherland. *The Architecture of the Remos System*, Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing, August 2001, California, USA.

[5] Jason Flinn, Dushyanth Narayanan and M. Satyanarayanan. *Self-Tuned Remote Execution for Pervasive Computing*, In Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII) May 2001, Schloss Elmau, 82493 Elmau/Oberbayern, Germany.

[6] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, L. Zhang. *IDMaps: A Global Internet Host Distance Estimation Service*, To appear in IEEE/ACM Trans. on Networking, Oct. 2001.

[7] GZIP. http://www.gzip.org/.

[8] Richard Han, Pravin Bhagwat, Richard LaMaire, Todd Mummert, Veronique Perret, and Jim Rubas. *Dynamic Adaptation in an Image Transcoding Proxy for Mobile Web Browsing*, IEEE Personal Communications Magazine, December 1998.

[9] Hemy, M., Hengartner, U., Steenkiste, P., and Gross, T.. *MPEG System Streams in Best-Effort Networks*, Proc. of PacketVideo '99, New York, April 1999.

[10] Michael Hemy, Peter Steenkiste, and Thomas Gross. *Evaluation of Adaptive Filtering of MPEG System Streams in IP Networks*, IEEE International Conference on Multimedia and Expo 2000, New York, New York.

[11] Van Jacobson. *pathchar - a tool to infer characteristics of Internet paths*, presented as April 97 MSRI talk.

[12] Kevin Lai and Mary Baker. *Nettimer: A Tool for Measuring Bottleneck Link Bandwidth*, Proceedings of the USENIX Symposium on Internet Technologies and Systems, March 2001.

[13] M. Kim and B. D. Noble. *Mobile network estimation*, in the Seventh ACM Conference on Mobile Computing and Networking, July 2001, Rome, Italy.

[14] C. Krintz and B. Calder. *Reducing Delay with Dynamic Selection of Compression Formats*, Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing, August 2001, California, USA.

[15] Nancy Miller and Peter Steenkiste. *Collecting Network Status Information for Network-Aware Applications*, Proceedings of INFOCOM 2000.

[16] Brian D. Noble. *Mobile Data Access*, Ph.D. Thesis, CMU-CS-98-118, May 11, 1998.

[17] Vern Paxon, Andrew Adams and Matt Mathis. *Experiences with NIMI*, In Proceedings of the Passive and Active Measurement Workshop, 2000.

[18] RFC 1157. *A Simple Network Management Protocol (SNMP).*

[19] RFC 1213. *Management Information Base for Network Management of TCP/IP-based internets: MIB-II.*

[20] M. Satyanarayanan. *Pervasive Computing: Vision and Challenges*, In IEEE Personal Communications, pp. 10-17, August, 2001.

[21] M. Satyanarayanan. *Mobile Information Access*, In IEEE Personal Communications, Vol. 3, No. 1, February 1996.

[22] Stefan Savage. *Sting: a TCP-based Network Measurement Tool*, Proceedings of the 1999 USENIX Symposium on Internet Technologies and Systems, pp. 71-79, Boulder, CO, October 1999.

[23] S. Seshan, M. Stemm, R. H. Katz. *SPAND: shared Passive Network Performance Discovery*, In Proc 1st Usenix Symposium on Internet Technologies and Systems (USITS '97) Monterey, CA December 1997.

[24] Peter Steenkiste. *Adaptation Models for Network-Aware Distributed Computations* 3rd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC'99), Orlando, January 8, 1999.

[25] Rich Wolski, Neil T. Spring, and Jim Hayes. *The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing*, Journal of Future Generation Computing Systems, 1999, also UCSD Technical Report Number TR-CS98-599, September, 1998.

[26] N. Yeadon, F. Garcia, D. Hutchison, and D. Shepherd. *Filters: Qos support Mechanisms for Multipeer Communications*, IEEE Journal on Selected Areas in Communications, 14(7):1245-1262, Sept 1996.

Implementation of a Recursive Function as a Split-Phase Abstract Machine

Suraj Sudhir
ssudhir@ece.cmu.edu
Carnegie Mellon University
Pittsburgh PA 15213

Abstract

This report describes the implementation of recursive functions as a split-phase abstract machine, a partitioning model that we use to implement programs in high-level languages directly on reconfigurable architectures. The fibonacci function is implemented as an example to demonstrate the technique used. As a description of an evolving work, this report outlines not just the implementation of a single programming paradigm but also broadly outlines how programs with different constructs and coding paradigms may be effectively translated to hardware and placed and routed efficiently on high-density reconfigurable devices

1 Introduction

The ability to directly compile a program in a high-level programming language into hardware implementation (i.e., the generation of *application specific hardware*) on a reconfigurable computing device is a promising technique to achieve significant speedups in computing performance, and therefore make general-purpose processors redundant. Such capability would permit a greater mainstream acceptance of the promise of reconfigurable computing, since programmers would not be required to re-implement code in a custom hardware definition language.

Most programming constructs have equivalent logical functions so that they can be directly translated to hardware. The if-then-else function, for example, is a simple true-false evaluation of an expression. Programming paradigms like recursion presents problems, however. This is because recursion is not a combinational logic function but a sequential function, with combinational logic as well as state information. This report examines a framework to identify such functions in code and automatically generate hardware that implements the code.

Further, this report introduces preliminary efforts on studying the problem of placement and routing of such an implemented code on a high-density reconfigurable array, such as a Nanofabric [2]. A theoretical framework that are relevant to this placement problem is described.

2 Background

Existing reconfigurable architectures have primarily been used for fast hardware implementation of custom hardware, embedded circuits, etc. They are however not dense enough to support fullscale program implementation in hardware. Nanoscale devices provide many orders of magnitude increase in the number of reconfigurable gates available.

Implementing programs on such devices requires a framework for partitioning the program into discrete units so that they can be efficiently placed on a dense reconfigurable device. This is because reconfigurable devices are invariably organized as a regular structure of configurable units.

In [2] the authors describe a Split-Phase Abstract Machine (SAM) that allows programs to be broken up into autonomous units. To do this, the compiler partitions the program at split-phase instructions, which are instructions of non-deterministic latency, e.g., memory accesses. Partitioning programs along split-phase instructions yields collections of instructions that we call split-phase threads (STs) which are implemented on the configurable units of the interconnected reconfigurable network.

Our implementation framework assumes a decentralized collection of STs so that a central control unit, which would be a configuration and communication bottleneck, is avoided. Each ST computes a result and based upon this result, 'fires' another ST.

3 Framework

3.1 Partitioning

A compiler-directed scheme to convert code to hardware needs to determine two things: the conditions under which control is transferred from one ST to another, and what constitutes the data that is transferred between STs.

The first task can be accomplished by determining whether the split-phase instruction that terminates the ST is a conditional instruction. If not, there is only one ST to which control can be transferred.

The second task is more complex. Three pieces of information are important. First, any computed result that is needed by subsequent STs needs to be saved. Also, all variables that are required by subsequent STs needs to be saved and propagated. Finally, any variable that is required by the same ST during a later invocation (e.g. if the ST is part of a loop) needs to be saved.

The compiler can determine this state information using liveness analysis. All data that is live at the split-phase instruction needs to be saved as state and propagated to the next ST being called. While implementing this scheme, an important distiction needs to be taken into account for the case of recursive functions. State information can normally be saved within registers and communicated to subsequent STs. However, if the split-phase instruction of the current ST happens to be a call to a recursive function, a register will not suffice for the purpose of storing this information.

This is because the recursive function may call itself multiple times. As a result, saving the last value of the live variables alone will not work. The history of their values during each successive invocation must be known. Therefore the state information, which corresponds to the activation record in a normal software implementation of recursion, needs to be preserved in stacks, not registers. The state information that is preserved at the end of STs can be differentiated as non-recursive or recursive state.

Therefore a compiler-directed approach would do the following:

- Split the program along the split-phase instructions and identify the STs.
- Use liveness information at the end of each ST to determine the state information associated with that ST.
- If the split-phase instruction is not a recursive function call, then annote the intermediate level code so that the compiler backend generates registers to hold this data. If the split-phase instruction happens to be a recursive function call, then the compiler backend should be instructed to generate stacks for each of the live variables at the end of the ST.

Todo: *Implement full compiler-directed scheme*

3.2 Placement

The collection of STs generated using the SAM paradigm forms a graph, with the nodes being the STs and the arcs being the control transfers between them. The reconfigurable device itself forms a two-dimensional rectangular array.

The placement problem thus becomes one of placing a directed graph on an orthogonal grid. [1] describes a linear-time solution to this theoretical problem. A useful theorem states that a 4-connected graph with n nodes can be placed in linear time on an nxn grid with at most 2n+2 bends in the arcs.

This is of great significance, since the number of nodes in the graph would be very large, and a linear time algorithm would be advantageous. Other works [4, 3] present heuristics that build upon the work presented in [1] in order to minimize the interconnect length, which would aid routing of the placed graph.

Todo: *Implement placement and routing algorithm. Study problem in the case of constrained orthogonal grids*

References

[1] Therese Biedl. Embedding nonplanar graphs in the rectangular grid. Technical Report RRR27-93, Rutgers University, 1993.

[2] Seth Copen Goldstein and Mihai Budiu. Nanofabrics: Spatial computing using molecular electronics. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, 2000.

[3] Gunnar W. Klau and Petra Mutzel. Optimal compaction of orthogonal grid drawings. In *Proceedings of Integer Programming and Combinatorial Optimization conference*, 1999.

[4] Achilleas Papakostas and Ioannis G. Tollis. Algorithms for area-efficient orthogonal drawings. *Computational Geometry. Theory and Applications*, 9(1-2):83–110, 1998.

Verifiable Secret Redistribution
(Extended Abstract)

Theodore M. Wong *†
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Suppose we have distributed shares of some secret information to a set of n servers such that we can reconstruct the secret, or perform distributed computations, with m of the n shares. Examples of secrets include objects in survivable storage systems [9] or multiparty signature keys [3, 4, 5, 6, 7]. If a server fails or is subverted by an adversary, we may wish to redistribute the remaining shares to a new set of n' servers. Since we do not trust servers with the secret itself, we wish to perform redistribution without reconstruction of the secret. We also wish to verify that the new shareholders have **valid** shares of the secret (ones that can be used to reconstruct the secret).

We present a new protocol to perform non-interactive **verifiable secret redistribution** (VSR) for secrets distributed with Shamir's secret sharing scheme [8]. Suppose we have distributed shares of a secret to shareholders in Shamir's (m, n) **threshold scheme** (one in which we require m of n shares to reconstruct the secret), and wish to redistribute the secret to shareholders in a new (m', n') threshold scheme. Furthermore, suppose we wish to avoid reconstruction of the secret. Our VSR protocol enables the redistribution of shares from the old to new shareholders without reconstruction of the secret by any of the shareholders, and guarantees that the new shareholders have valid shares. Our protocol guards against faulty behavior by up to $n - m$ of the old shareholders provided that $m > \frac{n}{2}$. Figure 1 illustrates the application of our VSR protocol.

We base our VSR protocol on Desmedt and Jajodia's general redistribution protocol for linear secret sharing schemes [1], which we specialize for Shamir's scheme. In their protocol, m of the n old shareholders each distribute n' **subshares** of their shares of a secret, and the n' new shareholders combine m subshares (one from each old shareholder) to create new shares of the secret. m' new shares are required to reconstruct the secret. Unlike our protocol, their protocol assumes non-faulty old shareholders. Thus, a faulty old shareholder, without the risk of detection, may cause the new shareholders to create **invalid** shares (ones that cannot be used to reconstruct the secret) by distributing invalid subshares.

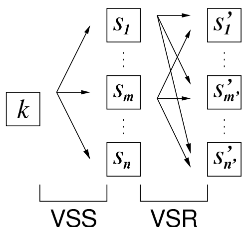


Figure 1: Initial distribution of a secret k with Shamir's (m, n) threshold secret sharing scheme, followed by redistribution to an (m', n') threshold scheme. Verifiable secret sharing (VSS) schemes can be used to guarantee that the shares $s_1 \dots s_n$ are valid. Our new verifiable secret redistribution (VSR) protocol can be used to guarantee that the shares $s'_1 \dots s'_{n'}$ are valid.

We extend Desmedt and Jajodia's redistribution protocol with Feldman's non-interactive **verifiable secret sharing** (VSS) scheme [2] to ensure that a SUBSHARES-VALID condition is true during redistribution. With Feldman's scheme, each old shareholder broadcasts a zero-knowledge proof of the validity of the subshares to the new shareholders. The new shareholders verify the proof without further interaction with the old shareholders. Feldman assumes there exist homomorphic encryption functions that are hard to invert, allowing the old shareholder to broadcast encryptions of their share and the subshare generation function without revealing them. Feldman also assumes there exist reliable broadcast communication channels among all participants and private channels between every pair of participants.

*This research is sponsored by the Defense Advanced Research Projects Agency (DARPA), Advanced Technology Office, under the title "Organically Assured and Survivable Information Systems (OASIS)" (Air Force Cooperative Agreement no. F30602-00-2-0523). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of DARPA or the U.S. Government.

†We would like to thank Michael Reiter, Chenxi Wang, and Jeannette Wing for their technical input and support.

We show that the SUBSHARES-VALID condition is necessary but not sufficient to guarantee that the new shareholders create valid shares, and present a new SHARES-VALID condition. The old shareholders broadcast a zero-knowledge proof of the validity of their shares of the secret to the new shareholders. As before, the new shareholders verify the proof without further interaction with the old shareholders. The check of the SHARES-VALID condition also assumes there exist homomorphic encryption functions that are hard to invert, allowing the old shareholders to prove the validity of their shares to the new shareholders without revealing them. We prove that the SUBSHARES-VALID and SHARES-VALID conditions are necessary and sufficient to guarantee that the new shareholders create valid shares of the original secret.

References

[1] DESMEDT, Y., AND JAJODIA, S. Redistributing secret shares to new access structures and its applications. Technical Report ISSE TR-97-01, George Mason University, Fairfax, VA, July 1997.

[2] FELDMAN, P. A practical scheme for non-interactive verifiable secret sharing. In *Proc. of the 28th IEEE Ann. Symp. on Foundations of Computer Science* (October 1987), IEEE, pp. 427–437.

[3] FRANKEL, Y., GEMMELL, P., MACKENZIE, P. D., AND YUNG, M. Optimal resilience proactive public-key cryptosystems. In *Proc. of the 38th IEEE Ann. Symp. on Foundations of Computer Science* (October 1997), IEEE, pp. 384–393.

[4] FRANKEL, Y., GEMMELL, P., MACKENZIE, P. D., AND YUNG, M. Proactive RSA. In *Proc. of CRYPTO 1997, the 17th Ann. Intl. Cryptology Conf.* (August 1997), B. S. Kaliski Jr, Ed., vol. 1294 of *Lecture Notes in Computer Science*, Intl. Assoc. for Cryptologic Research, Springer-Verlag, pp. 440–454.

[5] GENNARO, R., JARECKI, S., KRAWCZYK, H., AND RABIN, T. Robust threshold DSS signatures. In *Proc. of EUROCRYPT 1996, the Intl. Conf. on the Theory and Application of Cryptographic Techniques* (May 1996), U. M. Maurer, Ed., vol. 1070 of *Lecture Notes in Computer Science*, Intl. Assoc. for Cryptologic Research, Springer-Verlag, pp. 354–371.

[6] HERZBERG, A., JAKOBSSON, M., JARECKI, S., KRAWCZYK, H., AND YUNG, M. Proactive public key and signature systems. In *Proc. of the 4th ACM Intl. Conf. on Computer and Communications Security* (April 1997), Assoc. for Computing Machinery, pp. 100–110.

[7] RABIN, T. A simplified approach to threshold and proactive RSA. In *Proc. of CRYPTO 1998, the 18th Ann. Intl. Cryptology Conf.* (August 1998), H. Krawczyk, Ed., vol. 1462 of *Lecture Notes in Computer Science*, Intl. Assoc. for Cryptologic Research, Springer-Verlag, pp. 89–104.

[8] SHAMIR, A. How to share a secret. *Communications of the ACM 22*, 11 (November 1979), 612–613.

[9] WYLIE, J. J., BAKKALOGLU, M., PANDURANGAN, V., BIGRIGG, M. W., OGUZ, S., TEW, K., WILLIAMS, C., GANGER, G. R., AND KHOSLA, P. K. Selecting the right data distribution scheme for a survivable storage system. Technical Report CMU-CS-01-120, Sch. of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 2001.