Program Interpolation

A. Moss
moss@cs.bris.ac.uk
University of Bristol

D. Page
page@cs.bris.ac.uk
University of Bristol

Abstract

Program interpolation is a new type of transformation that given an input program written in a specially constructed Domain Specific Language (DSL), produces a family of functionally equivalent instruction sequences as output. Each sequence is an "interpolation" between the control-flows of implementation strategies supplied in the input program. The purpose of the transformation is to expose behavioural differences (e.g. performance) within the sequences, and thus allow automated optimisation with respect to architectural trade-offs that are difficult to quantify and model. We present results from a prototype compiler that demonstrate a 63% speedup in the domain of multi-precision integer arithmetic.

1. Introduction

The decision to use a low-level programming language for a given implementation task is typically guided by performance goals; this is despite the fact that such a selection can present significant challenges to a programmer. In short, such programmers are required to write programs that bridge a gap between high-level algorithms and implementation strategies (and choices thereof), and the complexities of processors that will execute the result. Mapping behaviour within a given algorithm onto the behavioural characteristics of the processor is often hard enough: to meet the demands for increased performance, processor design has broken the normal sequential computational model apart and rebuilt a sophisticated parallel processing model that looks functionally equivalent to the programmer but has radically different behaviour. Underneath this model, aggressive use of concepts such as out-of-order execution, superscalar dispatch and memory caches has resulted in a behavioural model of incredible complexity. However, this is exacerbated by subtle trade-offs between different algorithmic options that provide a "moving target" which is hard to resolve in programs written at a low-level. These trade-offs can be resolved to some extent by asymptotic analysis of the high-level algorithms, but the aforementioned complexity of any given target platform can have a significant impact on any constant factors and, in some cases, reduce the worth of such analysis in reality.

By identifying a problem domain in which low-level programming is both an accepted and necessary art, we aim to attack the explosion in design space. That is, by introducing a novel language design and program transformation strategy which we call *program interpolation*, we aim to mitigate the problem of selecting between a large number of algorithmic and implementation options based on trade-offs between them which are often too subtle for human programmers to capitalise on. The basic idea is to provide a language which is tightly coupled to the problem domain and includes some unusual features such as non-deterministic choice and a split-level language design (using a functional part and imperative part). Using this language, the programmer provides an input program composed of many functionally equivalent clauses which roughly detail the various implementation options. This input program is fed to a compiler which computes blended programs that lie "between" the options. The process which searches through such "interpolations" uses dynamic feedback from real measurements on the target platform. As a result, the system can automatically construct implementations that capitalise on the behavioural model of the processor without the programmer needing to understand this model or, crucially, to explore the design space manually.

Public-key cryptography is currently dominated by two schemes for securing information whether in transit or in storage. RSA [18] is based on arithmetic in the ring $\mathbb{Z}_N$, where $N = p \cdot q$ for large primes p and q, while Elliptic Curve Cryptography (ECC) can be parametrised over the finite field $\mathbb{F}_p$ for large prime p. In both RSA and ECC, computation is dominated by the cost of performing modular arithmetic in the appropriate structure, currently 2048-bits and 256-bits respectively. Performance is dominated by multi-precision integer arithmetic. The algorithms are well understood [15, Chapter 14], but subtle variations in implementation are highly influential on resulting performance. Bernstein's [1] survey of efficient multiplication gives a good overview of this topic.

On processors with a w-bit word size, it is common to represent operands using a vector of n radix-2^w digits, i.e. n machine words. Asymptotic analysis groups multiplication algorithms into the $O(n^2)$ complexity of school-book algorithms, the $O(n^{\log_2 3})$ complexity of Karatsuba-Ofman [12] multiplication; or the $O(n \log_2 n)$ complexity of FFT-based multiplication. The differing constant factors mean that the more complex algorithms are quicker on smaller operand sizes, under 500-bits in size for example, school-book multiplication will typically be fastest; the exact break-even is dependent upon the particular processor.

Once a particular algorithm is choosen a further set of options is presented for the exact implementation. As an example, imagine selecting school-book multiplication. Given the vector representation of operands x and y, one alternative is to scan through the digits of y in turn, performing $x \cdot y_i$ and accumulating the results. This method is known as operand-scanning and naively requires four addition operations for each multiplication. Alternatively, product-scanning (also known as Coomba [5] multiplication), processes the individual products $x_i \cdot y_j$ in ascending order of the expression $i+j$. Efficient operand-scanning reduces the number of addition operations to $2 \cdot n^2$ for the n^2 partial products. The method requires a number of registers linear in the problem size to avoid extra memory accesses. By contrast product-scanning uses $3 \cdot n^2$ additions, but only requires three registers. Selection between the two options

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]...$5.00

depends on a fine balance between register pressure and latency of memory access on the target platform; this trade-off is subtle as a result of hidden complexity within most concrete processors.

Few areas in Computer Science still require low-level assembly programming for acceptable performance, yet for cryptography it is vital. Security is an invisible feature: users only notice when it is broken. Thus the performance must be high enough that the user does not notice the overhead. Processor features that aide the programmer are typically not exposed by high level languages. For example, processor instruction sets typically include carry-flags and an operation for w-bit $\times w$-bit multiplication onto a $2w$-bit result. Without inline assembly the missing bits must be explicitly computed, impacting performance. Languages with native multi-precision arithmetic offer access to generic library routines, rather than the low-level operations to implement specialised arithmetic. The difference in performance is often an order of magnitude. Low-level assembly is currently necessary to exploit the subtle implementation changes that can maximise performance.

Existing Program Transformation techniques have been applied to this problem domain. Standardisation of security parameters means one can aggressively specialise an implementation for given n and w; an exemplar of this approach is given by the $\mathtt{mp}\mathbb{F}_q$ system of Gaudry and Thomé [7].

Gura et al. focus on the use of school-book multiplication to implement ECC on highly constrained embedded platforms. They approach the problem differently by describing a parametrised algorithm [8]. Operand-scanning is used for inner loops, controlled by a product-scanning outer loop. Tuning a parameter allows selection of a hybrid strategy between the two extremes. The key insight of Gura et al. is that such a hybrid allows a closer fit to the characteristics of a particular processor: they demonstrate that suitable tuning of the parameters produces a higher performance than either approach alone. Scott and Szczechowiak [19] and Uhsadel et al. [20] recently expanded on this method for particular processors.

Although this approach is attractive it places more burden on the programmer. Determining the exact parameterisation to tune the hybrid algorithm to a particular platform is left as a problem for the programmer to solve. Conventional languages lack a suitable mechanism to describe this problem to the compiler. This calibration problem is similar to that faced by libraries such as GMP, BLAS or FFTW. These libraries all employ a calibration procedure that uses dynamic feedback from the installation platform to determine which choice of implementations is most suitable.

Our work is motivated by the observation that the approach of Gura et al. is attractive but would benefit from language support for automatic calibration. We propose a new style of language that allows the programmer to specify alternative implementations. In this paper we present a Domain Specific Language for multi-precision arithmetic, and preliminary results from our prototype compiler. Our results are suprising because they show that even the most simple problem in the domain can be tuned to take advantage of the complex architectural tradeoffs in a real-world processor. The burden on the programmer is reduced by automating the implementation selection; instead of calibrating different design choices it is only necessary to specify them. These results represent preliminary work to establish the validity of the approach; we are optimising a simple program for a processor with a complex performance model.

2. Motivating Example

Consider an example problem: given m input variables, each of which is a multi-precision integer represented by a vector of n radix-2^w digits, compute the sum of all m variables. Although a simplification of our problem domain, this example is a model of the scheduling of additions to sum the partial products within a school-book multiplication. By investigating this problem in detail,

```
// partitioning step digit-wise
SUM( vl, vh, dl, dh : dl < d < dh - 1 ) {
  SUM( vl, vh, dl,    d + 0 );
  SUM( vl, vh, d + 1, dh    );
}

// partitioning step variable-wise
SUM( vl, vh, dl, dh : vl < v < vh - 1 ) {
  SUM( vl,    v + 0, dl, dh );
  SUM( v + 1, vh,    dl, dh );
}

// digit-wise pure strategy
SUM( vl, vh, dl, dh ) {
  SET t = 0;
  for i in dl to dh {
    for j in vl to vh {
      ADS x[ j, i ], r[ i ], t, r[ i ];
    }
    SET r[ i + 1 ] = t;
  }
}

// variable-wise pure strategy
SUM( vl, vh, dl, dh : dl < dh ) {
  for j in vl to vh {
    ADD x[ j, dl ], r[ dl ], r[ dl ]
    for i in dl + 1 to dh - 1 {
      ADC x[ j, i ], r[ i ], r[ i ]
    }
    ADCS x[ j, dh ], r[ dh ], r[ dh + 1 ], r[ dh ];
  }
}
```

Figure 1. An example program written in our DSL that computes the sum of m vectors, each of n digits. The input and output vectors `x` and `r` are implicitly defined and globally accessible.

our aims are two-fold: to provide a motivating example within the space constraints of a single research paper, and to demonstrate that a performance advantage can be achieved even for a problem that does not appear to warrant much optimisation effort.

Figure 1 contains a DSL program that computes the example problem. Section 3 will describe the grammar and semantics of the DSL; here we describe the intuition of how the program works. Programs in the DSL consist of a set of procedures. Each clause contained a possible implementation for the named procedure. The list of arguments to each clause capture input values, and can be guarded by a contraint. The order of clauses is not significant; when multiple clauses can satisfy a recursive call a non-deterministic choice is made between them. The optional constraint allows the programmer to control when clauses can overlap and multiple execution paths are valid. A program with multiple possible execution paths can be expanded into a family of deterministic implementations. By choosing a particular implementation the compiler is specialising away the non-deterministic construct.

The `SUM` procedure computes the sum of a region in the problem, specified as a range of vectors `vl`...`vh` and a range of digits `dl`...`dh`. For every region a non-deterministic choice is made: either divide the region into subregions and recursively compute their sums, propagating the final carry between regions, or apply one of the pure strategies to the entire region. When the partitioning clauses are selected a further source of non-determinism arises. Consider the first clause, the constraint `dl<d<dh-1` contains three

parameters. While `dl` and `dh` are bound by the arguments passed to the call, `d` is allowed to take any legal value in the range.

The two pure strategies roughly corrospond to operand-scanning or product-scanning in a multiplication. *Digit-wise* evaluation computes the sum of each digit in turn by accumulating the corresponding digit from each vector. Given m vectors, the region produces a $\log_2(m)$-bit final carry. *Variable-wise* evaluation accumulates each vector in turn, by accumulating digits onto the output with a running carry, and producing a 1-bit final carry.

The important tradeoff between the two strategies is between the number of addition operations and the amount of memory traffic. The traffic is determined by how many registers are required, as if this is greater than the number on the target architecture spills must be generated. Both strategies appear to use $m \cdot n$ additions. The instruction semantics are explained in Section 3.2, but broadly speaking `ADS` and `ADC` differ in cost. Only `r[i]` and `t` are live in the digit-wise strategy so only a constant number of registers are required. The variable-wise strategy uses n elements of `r`, so a linear number of registers are required.

The performance implications for selection of one of the pure or hybrid strategies is far from clear. Trade-offs involving register pressure, instruction latency and memory accesses are difficult to quantify and model. As clear performance models are not available for modern processor architectures it is not possible to determine the best strategy by inspection. Once hybrid strategies have been identified, it is troublesome to write a separate program to investigate each case. Cleary automating this process improves programmer productivity. The purpose of *program interpolation* is to delegate optimisation decisions to the compiler, which is able to explore the design space automatically as a result of the program description. The goal is to determine if one of the many interpolations between the supplied strategies can produce a more efficient result than the initial input.

3. Language

Our DSL has been designed to allow the programmer to express several variants of an algorithm, which represent different implementation strategies, to the compiler. It is important that these variants allow enough low-level control to be competitive with implementation directly in assembly language. While it is desirable to allow the programmer to guide placement of individual instructions, the DSL is designed so that the result is reasonably portable. Our approach is influenced by the combination of two previous results.

Bernstein [2] describes `qhasm`, a low-level assembly language which has been enhanced with a register allocator. The programmer still writes code in the assembly language but can use variables rather than registers; the result is conceptually similar to writing inline assembly language within a C program. By performing register allocation, the `qhasm` assembler demonstrates that while programmers might insist on using low-level languages, it is still possible to automate more mechanical aspects of their workload.

Moss and Muller [16] embed the semantics of an assembly language within a Prolog-based meta-language. Their experiment relies on the programmer to ensure that all Prolog control-flow is static and can be specialised from the program to leave a pure assembly language residual.

Both approaches are interesting experiments in how low-level languages can be improved with selected high-level features. We have borrowed the main ideas from each, and combine them into our DSL which is a composition of two parts: an "inner" assembly language which is embedded into an "outer" meta-language.

Moss and Muller [16] used a simple micro-controller as their target. The fast and uniform access to the accumulator register and memory locations in the device simplified details of the embedding. Our target processors are significantly more complex and rely on efficient use of the register set to avoid relatively slow access to memory. In order to maintain a simple embedding we define a "thin" layer of Virtual Machine (VM) that is designed specifically for our problem domain, rather than use the instruction set directly.

Each VM instruction may expand to become multiple native instructions depending on both the target architecture and the specific register allocation at that point in the execution. This delayed register allocation, similar in spirit to `qhasm`, allows the VM to be treated as an ∞-register machine. Register allocation does not introduce any additional control-flow or data-dependencies beyond those in the VM instruction sequence. For this reason the VM instructions are a simple model of the target processor, but analyses performed on the VM instruction sequence are sound; this type of mapping is called a *syntactic isomorphism* [16].

3.1 Bindings and State

The DSL is designed with four types of variables to cleanly separate mutable state from functional bindings. The separation is used in the language semantics to prevent run-time data from affecting control-flow and to ensure that each program can be specialised into straight-line code. Our problem domain consists of programs that possess this property, but which are conventionally written in languages that do not enforce it.

Temporary variables are single w-bit words with scope local to a single procedure call. Each variable accessed by a simple (non-indexed) textual label.

Vector variables contain a constant number of w-bit digits. Each variable has global scope within the program and is referred to via the form `v[i]` where v is a textual label and i is an index.

Table variables are a two-dimensional collection of w-bit digits. Each has global scope and is referred to via the form $\mathtt{v}[\mathtt{j},\mathtt{i}]$ where j and i are indices to indicate the row and column respectively.

Indices are a map from labels onto integers with local scope.

Program inputs are received and outputs returned via nominated variables. The example in Figure 1 uses the table `x` defined to be n by m digits as input, and the vector `r` defined as $n+1$ digits. A parametrisation of the example for a specific problem size is given as an initial call e.g. `SUM( 0,3, 0,5 )` in the case that n is 4 and m is 6. The parametrisation being compiled fixes the size of the variables for a particular execution.

The index binding is a function f over labels l initially defined as $f(l) = \bot$. The function f is always evaluated at compile-time, and use of an undefined label generates an error. The binding can be updated on a procedure call to monotonically increase the set of bound labels. Update of the binding f to map the label l onto the integer c is defined as:

$$f[l \leftarrow c](x) = \begin{cases} v & \text{if } x = l \\ f(x) & \text{otherwise} \end{cases}$$

Simple expressions over indices are allowed within the language, permitting integer operations over both constants and bound labels:

$$\begin{aligned} f[\![l]\!] &= f(l) \\ f[\![c]\!] &= c \\ f[\![e_1 + e_2]\!] &= f[\![e_1]\!] + [\![e_2]\!] \end{aligned}$$

Any well-defined integer operation can be considered within the index expressions; the evaluation of expressions occurs at compile-time. Only additive operations are required in the example.

Separate to the index binding f is a mutable state $\mathbb{V}$ that contains the values of temporaries, vector and table variables. We also consider the state to contain a distinguished 1-bit variable CF that represents the carry-flag. Note that after parsing the input program,

$$
\begin{aligned}
\textsc{In}(<\texttt{ADD a,b,r}>,\ f,\ \mathbb{V}) &= \mathbb{V}\left[\textsc{Op}(\texttt{r}) \leftarrow \mathbb{V}(\texttt{a}) + \mathbb{V}(\texttt{b}) \bmod 2^w,\ CF \leftarrow \left\lfloor \frac{\mathbb{V}(\texttt{a}) + \mathbb{V}(\texttt{b})}{2^w} \right\rfloor\right] \\
\textsc{In}(<\texttt{ADC a,b,r}>,\ f,\ \mathbb{V}) &= \mathbb{V}\left[\textsc{Op}(\texttt{r}) \leftarrow \mathbb{V}(\texttt{a}) + \mathbb{V}(\texttt{b}) + \mathbb{V}(\texttt{CF}) \bmod 2^w,\ CF \leftarrow \left\lfloor \frac{\mathbb{V}(\texttt{a}) + \mathbb{V}(\texttt{b}) + \mathbb{V}(\texttt{CF})}{2^w} \right\rfloor\right] \\
\textsc{In}(<\texttt{ADS a,b,c,r}>,\ f,\ \mathbb{V}) &= \mathbb{V}\left[\textsc{Op}(\texttt{r}) \leftarrow \mathbb{V}(\texttt{a}) + \mathbb{V}(\texttt{b}) \bmod 2^w,\ \textsc{Op}(\texttt{c}) \leftarrow \mathbb{V}(\texttt{c}) + \left\lfloor \frac{\mathbb{V}(\texttt{a}) + \mathbb{V}(\texttt{b})}{2^w} \right\rfloor,\ CF \leftarrow \perp\right] \\
\textsc{In}(<\texttt{ADCS a,b,c,r}>,\ f,\ \mathbb{V}) &= \mathbb{V}\left[\textsc{Op}(\texttt{r}) \leftarrow \mathbb{V}(\texttt{a}) + \mathbb{V}(\texttt{b}) + \mathbb{V}(\texttt{CF}) \bmod 2^w,\ \textsc{Op}(\texttt{c}) \leftarrow \mathbb{V}(\texttt{c}) + \left\lfloor \frac{\mathbb{V}(\texttt{a}) + \mathbb{V}(\texttt{b}) + \mathbb{V}(\texttt{CF})}{2^w} \right\rfloor,\ CF \leftarrow \perp\right] \\
\textsc{In}(<\texttt{SET a,r}>),\ f,\ \mathbb{V}) &= \mathbb{V}\left[\textsc{Op}(\texttt{r}) \leftarrow \mathbb{V}(\texttt{a})\right]
\end{aligned}
$$

Figure 2. Semantics of the VM instruction set.

many details become fixed compile-time constants: the values of w and n, valid index ranges for each variable, and the total number of temporary variables are all known at this point. For a given input program the mutable state is defined as a set of bindings to values:

$$
\begin{aligned}
\mathbb{V} &= \{\textsc{Var}(l, j, i) := c \mid \textsc{Valid}(l, j, i)\} \\
&\cup \ \{\textsc{Tmp}(l) := c \mid \textsc{Valid}(l)\} \\
&\cup \ \{CF := c\}
\end{aligned}
$$

where VALID is a function that defines the range of valid indices according to the parsed program and parametrisation. For simplicity the vector variable element `l[i]` is represented by $\textsc{Var}(l, 0, i)$. Look-up of the variable x within the state is denoted:

$$\mathbb{V}(\texttt{x}) = y \ \text{ s.t. } \ (x := y) \in \mathbb{V}$$

State mutations are used within the inner language to define the operation of instructions upon the state $\mathbb{V}$. We denote the mutation of an element x in $\mathbb{V}$ from the value y to the value y' as:

$$\mathbb{V}[x \leftarrow y'] = \mathbb{V} \setminus \{x := y\} \cup \ \{x := y'\}$$

Where an operation in the language requires a sequence of mutations $x_1 \leftarrow y_1, \ldots, x_k \leftarrow y_k$, and each x_i is distinct we will denote this sequence of operations $\mathbb{V}[x_1 \leftarrow y_1, \ldots, x_k \leftarrow y_k]$.

3.2 The "Inner" Language

Using this notation we define the VM as an ∞-register machine, where each w-bit element of $\mathbb{V}$ occupies a unique register. The operation of each instruction is defined as mutations over this state in Figure 2. Syntactically, each instruction is composed of a mnemonic and a set of operands. The operands in each instruction may refer to temporary, vector or table variables. For vector or table variables the indexing expression may contain elements of the binding f, to map these syntactic names onto elements of $\mathbb{V}$ we define the operands as:

$$
\begin{aligned}
&\textsc{Op}(\texttt{l}) && = \textsc{Tmp}(l) && \text{when } \textsc{Valid}(l) \\
&\textsc{Op}(\texttt{l[i]}) && = \textsc{Var}(l, 0, f[\![i]\!]) && \text{when } \textsc{Valid}(l, 0, f[\![i]\!]) \\
&\textsc{Op}(\texttt{l[j,i]}) && = \textsc{Var}(l, f[\![j]\!], f[\![i]\!]) && \text{when } \textsc{Valid}(l, f[\![j]\!], f[\![i]\!])
\end{aligned}
$$

It is clear from inspection of this definition of OP that evaluation of the binding f can be performed in advance so that the selection of which registers the VM operates on is constant. Where it is clear from the context that an operand `x` is a syntactic element we abuse notation slightly in Figure 2 to redefine state look-up as:

$$\mathbb{V}(\texttt{x}) = y \ \text{ s.t. } \ (\textsc{Op}(\texttt{x}) := y) \in \mathbb{V}$$

The first four instructions detail different methods for adding two w-bit operands to produce a $(w+1)$-bit result.

The instructions `ADD` and `ADC` are native to most instruction set. They perform addition that either uses, or does not use, the current carry flag respectively. The extra bit produced in the operation sets the carry flag. The instructions `ADS` and `ADCS` do not exist on most processors, but are a natural expression of the example problem. Rather than set the carry flag (and thus create positional constraints during instruction scheduling) the most significant bit produced by the addition is "spilled" into a machine word; the spilled bit is accumulated in a target variable. The state of CF is destroyed during this operation.

A straight-line sequence of VM instructions can be mapped directly to a sequence of native instructions, expanding the macro-instructions into appropriate sequences. To simplify the programmers task, the VM instructions operated on operands that are elements of $\mathbb{V}$. However, the native instruction operands need to be encoded either as memory references or registers; load and store operations need to be introduced so that the registers contain values as needed, and results are flushed back to memory.

This process of *register allocation* is a well studied problem within compiler design. We note that the instruction sequences under investigation always produce chordal interference graphs; recent results have shown that this simplifies the allocation problem considerably [9, 4, 3]. The specific result of register allocation used will affect the performance of each sequence. The exact effects are hard to model across all of the sequences in a particular family but one benefit of interpolation is that such analysis is not necessary: the interpolation requires the allocation to preserve the functional equivalence when mapping sequences from the VM to the target, but behavioural differences are tolerated and explored during the search process described in Section 4.

An advantage of this VM is that we can define the semantics of instructions to permit a clean analysis of the code, allowing a similar level of portability to languages such as C−− [17], and making low-level implementation easier for the programmer. As an example, most platforms have support for addition; depending on whether 3-address or 2-address instructions are used, this will either be standard addition or accumulation. Most processors also support a hardware carry-flag, and instructions to access it. However, these features do not typically map directly onto the needs of a given algorithm: a programmer must invent ways to preserve the mutable carry-flag between instructions without perturbing other data. By using the VM, the programmer is presented with a more uniform target: these complications are abstracted away just enough that they can be ignored but not so much that the gap between VM and target processor is too wide.

3.3 The "Outer" Language

Because there are no control-flow constructs within the inner language, writing non-trivial programs is impossible; each instruction is simply a single modification of the VM state. However, the algorithms being implemented use standard constructs roughly equivalent to C statement sequences, `if` conditionals and `for` loops. In or-

<NAME>	:=	an identifier called a	$\longrightarrow$	[<NAME>, a]
<COMP>	:=	a simple comparison expression called a	$\longrightarrow$	[<COMP>, a]
<EXPR>	:=	a simple arithmetic expression called a	$\longrightarrow$	[<EXPR>, a]
<INST>	:=	a VM instruction called a	$\longrightarrow$	[<INST>, a]
<ARGS>	:=	a : <NAME> (‘,’ a : <NAME>)*	$\longrightarrow$	[<ARGS>, $a_0, a_1, \ldots a_{\lvert a \rvert-1}$]
<BLCK>	:=	‘{’ a : <STMT> + ‘}’	$\longrightarrow$	[<BLCK>, $a_0, a_1, \ldots a_{\lvert a \rvert-1}$]
<PROG>	:=	a : <PROC>+	$\longrightarrow$	[<PROG>, $a_0, a_1, \ldots a_{\lvert a \rvert-1}$]
<PROC>	:=	a : <NAME> ‘(’ b : <ARGS> (‘:’ c : <COMP>)? ‘)’ d : <BLCK>	$\longrightarrow$	[<PROC>, a, b, c, d]
<STMT>	:=	‘for’ a : <NAME> ‘in’ b : <EXPR> ‘to’ c : <EXPR> d : <BLCK>	$\longrightarrow$	[<LOOP>, a, b, c, d]
	|	a : <INST> ‘;’	$\longrightarrow$	[<INST>, a]
	|	a : <NAME> ‘(’ ‘:’ b : <ARGS> ‘)’ ‘;’	$\longrightarrow$	[<CALL>, a, b]

Figure 3. A grammar and associated AST rewrite rules for the outer language. The notation x : <RULE> associates a label x to some grammar rule <RULE>. As a result, one can view the <BLCK> rule, for example, as building a non-empty list called a containing <STMT> nodes. This list is packaged inside an AST node represented by another list headed by a token which determines the node type.

der to bridge the gap, a program written in the inner language is embedded into a program written in an outer meta-language. Our aim is to provide an apparently familiar set of control-flow constructs to the programmer. Careful restriction of the interaction between the two languages ensures a clean separation of state mutation and control-flow. This restriction guarantees that compile-time specialisation of the program will remove all control-flow constructs and leave only sequences of inner language instructions as a residual.

The outer language essentially performs macro expansion by defining the set of possible sequences of VM operations. No mutable variables exist in the outer language, the evaluation semantics are purely functional. If the outer language used no variables at all, then the variable references within VM instructions would be constant; the indexing provided in the outer language is a form of "syntactic sugar" to allow the programmer to define programs in a more convenient manner.

The syntax of the outer language is roughly defined by BNF-style grammar in Figure 3; this includes rewrite rules into Abstract Syntax Tree (AST) nodes which, for clarity, we work with rather than use the grammar rules directly. Notice that the inner language is embedded in the grammar of the outer language as the non-terminal <INST>; this combination defines the DSL the programmer uses. The structure of a program <PROG> in the DSL is a list of clauses; each clause <PROC> is similar to a procedure definition with a name and a list of arguments. This is added to by an (optional) constraint defined over the arguments to the clause; this allows, for example, for checks that an argument is within a given range. Recalling our example in Figure 1, we see that the program has four definitions that all define bodies for the clause SUM; the last two clauses act as recursive base-cases. The constraints included in the first two clauses prevent the associated bodies from operating on arguments outside a valid range. Each clause body, uses three main constructs: sequencing, contant bounded loops and non-deterministic choice.

The first two constructs allow the programmer to express standard iterative operations over vector operands in the algorithm. Since the loop bounds are a compile-time constant (as a result of fixed security parameters), the the programs will be equivalent to straight-line code after specialisation has (partially) unrolled the loops to suit the target platform.

Investigation of a split-level language using these two constructs was the focus in Moss and Muller [16]; the novelty of this work is the introduction of the third construct which forms the basis of the interpolation transformation. When the programmer uses the choice construct they indicate to the compiler that selection of either control-flow will result in a valid program, i.e. the construct is a guarantee from the programmer that the compiler can trust either version of the program to produce the same result. The compiler can specialise this construct at compile-time, creating a family of equivalent programs, each one taking a different decision.

The semantics of the language reflect the desire to separate the inner from the outer language. The inner language is imperative operating over $\mathbb{V}$; the outer language is a strict functional language defining the sequencing of elements within the inner language. To ensure efficiency it is important that specialisation should remove the outer language entirely leaving only a straight-line sequence of operations in the inner language. We discuss this guarantee more rigorously in Section 3.5 as a notion of program equivalence.

3.4 Control-Flow Semantics

There are two important mappings that must be defined on programs written in the DSL:

1. Each program maps an input VM state $\mathbb{V}$ onto an output state $\mathbb{V}'$; intuitively, this is the obvious result of the semantics of the inner and outer language parts.
2. Each possible execution of such a program maps an input state $\mathbb{V}$ onto a set of possible execution traces; each trace is a list that captures the VM instructions that were executed. The set of traces for any program is defined as the largest set of traces possible. That is, each non-deterministic choice in the program produces a set of traces with a common prefix (i.e. execution until that point) followed by every possible sequence of operations that the program defines afterwards.

The set of choices at each control point is independent of the current VM state; the clause constraints that define which bodies can be executed are defined over the set of bindings in the outer language (i.e. loop indices). Thus, the set of traces can be constructed without evaluation of the VM instructions. It is this property that ensures a clean separation between the inner and outer languages: sets of traces can be constructed and analysed without interpreting the inner operations or, conversely, individual traces can be executed without reference to the outer program that defined them. For our chosen domain we know that this is possible because each algorithm is parametrised by some function of the security parameter; although each algorithm is defined generically over any possi-

instruction: $$\text{EVAL}\Big(\llbracket \texttt{<INST>}, a \rrbracket,\ \mathbb{V},\ f\Big) = \text{IN}(a, f, \mathbb{V})$$

sequence: $$\text{EVAL}\Big(\llbracket \texttt{<BLCK>}, a_0 \rrbracket,\ \mathbb{V},\ f\Big) = \text{EVAL}\Big(\llbracket a_0 \rrbracket,\ \mathbb{V},\ f\Big)$$

sequence: $$\text{EVAL}\Big(\llbracket \texttt{<BLCK>}, a_0, \ldots, a_l \rrbracket,\ \mathbb{V},\ f\Big) = \text{EVAL}\Big(\llbracket \texttt{<BLCK>}, a_1, \ldots, a_l \rrbracket,\ \mathbb{V}',\ f\Big) \quad \text{s.t } \mathbb{V}' = \text{EVAL}\Big(\llbracket a_0 \rrbracket,\ \mathbb{V},\ f\Big)$$

loop: $$\text{EVAL}\Big(\llbracket \texttt{<LOOP>}, a, b, c, S \rrbracket,\ \mathbb{V},\ f\Big) \text{ s.t } b = c \quad = \text{EVAL}\Big(\llbracket S \rrbracket,\ \mathbb{V},\ f[a \leftarrow b]\Big)$$

loop: $$\text{EVAL}\Big(\llbracket \texttt{<LOOP>}, a, b, c, S \rrbracket,\ \mathbb{V},\ f\Big) \text{ s.t } b < c \quad = \text{EVAL}\Big(\llbracket \texttt{<LOOP>}, a, b+1, c, S \rrbracket,\ \mathbb{V}',\ f\Big) \quad \text{s.t } \mathbb{V}' = \text{EVAL}\Big(\llbracket S \rrbracket,\ \mathbb{V},\ f[a \leftarrow b]\Big)$$

call: $$\text{EVAL}\Big(\llbracket \texttt{<CALL>}, N, \overline{v} \rrbracket,\ \mathbb{V},\ f\Big) = \text{EVAL}\Big(\llbracket S \rrbracket,\ \mathbb{V},\ f'\Big) \quad \text{s.t} \quad \begin{array}{l} S \in \text{LOOK-UP}(N, \overline{v}, f) \\ f' = \text{BIND}(N, f, \overline{a}) \end{array}$$

Figure 4. Outer language semantics defined as the VM state produced by executing a DSL program represented as AST nodes.

ble security parameter, a particular instantiation will use constant-bounded loops and be equivalent to straight-line code.

Given that a statement S has the meaning $\llbracket S \rrbracket$, we define the evaluation function EVAL over AST nodes, the VM state and the index binding function. Hence the expression $\text{EVAL}(\llbracket S \rrbracket, \mathbb{V}, f)$ denotes the state produced by executing the statement S on state $\mathbb{V}$ with binding f. Using this notation we define the semantics of the outer language in Figure 4. The semantics of the inner language are embedded in the outer language using the function IN. This function applies the index binding to any indices within the operands of a, and then executes a on $\mathbb{V}$ according to the definitions in Section 3.2, returning the mutated VM state.

The definitions for sequencing and looping are straight-forward. It is important to note that while mutations of the VM state are explicitly passed through the control constructs, updates to the index binding can only occur locally within recursive evaluations of EVAL and cannot be passed along the flow of control. The index bindings are updated in two points; binding the current loop iteration to the index variable in order to evaluate the loop body, and binding values to procedure arguments.

The description of the procedure call mechanism relies on extra auxiliary functions that require further discussion. Once the input program has been parsed it is represented as a set $\mathbb{C}$ of 4-tuples, each of which represents a clause; the tuple contains a name, a list of arguments, a list of constraints and the clause body. When a call is encountered in the execution of a program it is necessary to decide which contexts control can pass to. Given a list of arguments $\overline{a}$ and a list of constraints $\overline{c}$ it is necessary to decide if the arguments satisfy the constraints; i.e. whether $\overline{a} \in \overline{c}$. Using the convention that the list of arguments contains tuples of labels and values, while the list of constraints contains tuples of values and domains we define satisfaction as:

$$\overline{a} \in \overline{c} \iff \forall (l, D) \in \overline{c} \ :\ \exists (l, v) \in \overline{a} \ :\ v \in D$$

To determine which clause bodies are eligible to receive control-flow we define a function LOOK-UP as mapping names and lists of values onto sets of statements. This is a standard device for introducing the mapping between a procedure name and its body statement. In this execution model it is valid for control to pass to multiple procedure bodies, and hence the set of statements that are returned. We define

$$\text{LOOK-UP}(N, \overline{v}) \quad = \quad \{\text{S} \mid \exists (N, \overline{a}, \overline{c}, S) \in \mathbb{C}\} \quad \text{s.t. } \text{ZIP}(\overline{a}, \overline{v}) \in \overline{c}$$

using the standard function ZIP to performs the mapping of elements in two lists, onto a list of tuples containing pairs of elements.

The function BIND updates the index binding with the argument values for a procedure call. Using standard notation that the head h and tail t of a list $\overline{a}$ are expressed $[h|t] = \overline{a}$ we define BIND as

$$\text{BIND}(N, f, \overline{a}) = \begin{cases} f & \text{if } \overline{a} = [] \\ \text{BIND}(N, f[l \leftarrow v], \overline{a}') & \text{when} [(l, v)|\overline{a}'] = \overline{a} \end{cases}$$

The result of executing the procedure call is a set of VM states. The central claim of the paper is that if the programmer has supplied a program that satisfies the notion of substitution equivalence (defined in Section 3.5) then this set of states will always be a singleton. Hence in the procedure call transition in Figure 4, any clause body S that results from a look-up will evaluate over the index binding to the same resultant state. The possible sequence of VM instructions executed may differ, and so in the next section we consider which sequences are produced by execution of the language elements.

Assuming the substitution equivalence on clauses in the program does not reduce the set of traces, and so there is not a simplification corresponding the semantic function above. The last property that we require from the language definition is that execution of any trace from the set produced will compute the correct state of the VM, i.e. the state defined by the semantic functions. To prove this formally we need to establish that the list of operations in each trace is a faithful recording of the operational steps that the semantic function undertakes.

The purpose of the split between the inner and the outer language is to produce the tight syntactic similarity between the trace function and the semantic function in the outer language. This similarity is possible because of the clean separation between state effects in the VM, and the control-flow of the program. Given the definitions of the trace function and the semantic function it follows naturally that every evaluation of program in the DSL corresponds to a sequence in the set of traces, and that every trace in the set computes the correct state.

Each element in the set of traces is a compilation of the DSL program into a straight-line program on the VM. Each program in the operations of the VM can be rewritten by a syntactic transformation to a program in the assembly language of the real target machine. Hence the trace function is a compiler for the DSL that produces a family of equivalent programs that correspond to the valid substitutions of control-flow within the source clauses specified by the recursion that the programmer has introduced. We term each such mixture of clause bodies an *interpolation* of the program.

$$\text{instruction:}\quad \textsc{Trace}\Big([\![<\texttt{Inst}>, a]\!],\ f\Big) = \{[a]\}$$

$$\text{sequence:}\quad \textsc{Trace}\Big([\![<\texttt{BLCK}>, a_0]\!],\ f\Big) = \textsc{Trace}\Big(a_0,\ f\Big)$$

$$\text{sequence:}\quad \textsc{Trace}\Big([\![<\texttt{BLCK}>, a_0, \ldots, a_l]\!],\ f\Big) = \Big\{x||y\ :\ x \in \textsc{Trace}\Big(a_0,\ f\Big),\ y \in \textsc{Trace}\Big([\![<\texttt{BLCK}>, a_1, \ldots, a_l]\!],\ f\Big)\Big\}$$

$$\text{loop:}\quad \underset{\text{s.t } b = c}{\textsc{Trace}\Big([\![<\texttt{LOOP}>, a, b, c, S]\!],\ f\Big)} = \textsc{Trace}\Big(S,\ f[a \leftarrow b]\Big)$$

$$\text{loop:}\quad \underset{\text{s.t } b < c}{\textsc{Trace}\Big([\![<\texttt{LOOP}>, a, b, c, S]\!],\ f\Big)} = \Big\{x||y\ :\ x \in \textsc{Trace}\Big(S,\ f[a \leftarrow b]\Big),\ y \in \textsc{Trace}\Big([\![<\texttt{LOOP}>, a, b+1, c, S]\!],\ f\Big)\Big\}$$

$$\text{call:}\quad \textsc{Trace}\Big([\![<\texttt{CALL}>, N, \overline{v}]\!], f\Big) = \Big\{\textsc{Trace}\Big([\![S]\!],\ f'\Big)\ :\ S \in \textsc{Look-Up}(N, \overline{v}, f),\quad f' = \textsc{Bind}(N, f, \overline{a})\Big\}$$

Figure 5. Definition of the TRACE function that computes the set of valid traces from an input DSL program represented as AST nodes.

3.5 Equivalence Definition

The DSL has a trivial termination property; all programs must terminate. However this property is only partly enforced by the language as a restriction on static loop bounds. The programmer must follow some informal restrictions when writing programs. Each recursive call must use parameters that are strictly smaller in some well defined ordering. In the example each call processes a region defined by $(\texttt{vl} \ldots \texttt{vh}, \texttt{dl} \ldots \texttt{dh})$, each recursive call partitions the region into smaller pieces ensuring that th recursion terminates. Restricting the DSL language to terminating programs simplifies notions of equivalence between programs. A standard notion of program equivalence

DEFINITION 1 (I/O Equivalence). *Programs P and Q are input-output equivalent if and only if for all inputs x the outputs $P(x)$ and $Q(x)$ are equal.*

is decidable on DSL programs following the informal restriction. Although in practice checking this relation may be intractable. When this equivalence holds between two programs, and they are merged together as clauses within a single program, it is not sufficient to guarantee that non-trivial interpolations exist. To ensure the existence of interpolations between clauses that produce behavioural differences they must compute the same input-output function in alternative ways. We capture this notion with a stronger equivalence relation. For these alternative computations to produce the same result the following conditions must hold:

- All clauses of a procedure compute the same result.
- All clauses of a procedure have the same set of valid inputs.

The first property is the standard notion of input-output equivalence defined over individual clauses rather than entire programs. This ensures that any non-deterministic selection of a clause body will produce the same values. The second property is more complex, it ensures that selection of a clause body can process the same set of inputs. It is possible that clauses may not be able to solve sub-problems because of some non-syntactic incompatibility. Consider a program `foo` that operates on its input by splitting into sub-problems. If the recursive clause could break into both odd and even sized sub-problems, but the base case could only process even sized problems then the clauses would be incompatible. Attempting to handle this compatibility property is a subtle, and thorny, issue at the language level. Although ensuring equivalence of clauses may be easier in the DSL we make the pragmatic assumption that the programmer guarantees these properties hold where necessary. The argument constraint system is provided to make this task easier.

DEFINITION 2 (Substitution Equivalence). *If a parsed program $\mathbb{C}$ contains two clauses $(N, \overline{a}_1, \overline{c}_1, S_1)$ and $(N, \overline{a}_2, \overline{c}_2, S_2)$ that share a label N and have overlapping constraints then for every argument vector $\overline{v}$ such that $\textsc{Zip}(\overline{a}_1, \overline{v}) \in \overline{c}_1$ and $\textsc{Zip}(\overline{a}_2, \overline{v}) \in \overline{c}_2$ then $\forall\, \mathbb{V}, f\ :\ \textsc{Eval}([\![S_1]\!], \mathbb{V}, f) = \textsc{Eval}([\![S_2]\!], \mathbb{V}, f)$.*

It is clear from the definition that substitution equivalence is a stronger notion than input-output equivalence. All programs that are substitution equivalent must compute the same terminal state for the outermost procedure call. Substitution equivalence is enforced at a finer granularity; procedure calls must be exchangeable without affecting the computed result. Rather than defining equivalence in isolation (over two entire programs) this new definition requires that fragments of control-flow (individual VM sequences) must be equivalent across a broader range of execution contexts.

This strengthening of the equivalence relation is analogous to the work of Lifschitz et. al [13] which defines strong equivalence over logic programs. The strengthening condition in that context is that logic programs P and Q must remain equivalent regardless of which facts or rules are added to the program database. The analogy that we draw with our own work is that our definition of equivalence strengthens the standard definition by allowing the replacement of entire clauses within the program, without perturbing the input-output characteristics.

3.6 Control-Flow Traces

Execution of a DSL program, as defined by the semantics of the EVAL function, performs a sequence of VM instructions. The non-deterministic choice of which procedure body to execute implies that execution of a single program can proceed in different ways, and hence follow many different sequences. A *trace* is a single sequence of VM instructions that is consistent with the EVAL semantics. In this section we construct the TRACE function that computes the set of traces that a program can execute. This set is defined as the largest set such that all elements are traces consistent with the program.

Figure 5 defines the TRACE function and hence the set of traces produced by a DSL program. As in the definition of EVAL the AST nodes produced from parsing the program as supplied as input, along with the index binding f. The VM state is not necessary as it does affect the sequence of instructions evaluated, only the state produced by them. So when the execution of a statement x precedes the execution of a statement y, there is a set of traces for the individual execution of x and y. The combined execution is a product of these traces, in that we must consider the concatenation of every prefix with every suffix:

Assuming the substitution equivalence on clauses in the program does not reduce the set of traces, and so there is not a simplification corresponding the semantic function above. The last property that we require from the language definition is that execution of any trace from the set produced will compute the correct state of the VM, i.e. the state defined by the semantic functions. A formal proof must show that each trace is a faithful recording of the operational steps in the semantic function.

The purpose of the split between the inner and the outer language is to produce the tight syntactic similarity between the trace function and the semantic function in the outer language. This similarity is possible because of the clean separation between state effects in the VM, and the control flow of the program. Given the definitions of the trace function and the semantic function it follows naturally that every evaluation of program in the DSL corresponds to a sequence in the set of traces, and that every trace in the set computes the correct state.

Each element in the set of traces is a sequence of VM operations executing the program. Each sequence can be rewritten by a transformation (including register allocation) into an assembly language program on the target platform. Hence the trace function is a compiler for the DSL; it produces a family of equivalent programs that correspond to valid substitutions of control-flow, within the source clauses specified. We term each such mixture of clause bodies an *interpolation* of the program.

4. Program Interpolation

Program optimisation attempts to improve program performance by transforming an input program into a functionally equivalent output program. A standard approach defines the transformation as a set of rewrite rules that restructure local sections of the program code. Rewrites can produce code sequences that enable further rules to be applied, or they can disable the application of certain rules. This approach can be considered to be an undirected search through a set of functionally equivalant programs.

Our approach is an attempt to try something fundamentally different. Instead of the programmer specifying a single program, and the optimiser trying to find equivalent programs through an undirected search, the interpolation language allows the programmer to specify an entire set of equivalent programs. The search problem for the interpolator then becomes a strategy to sample this set. The longterm aim is that the structure of this search space may be ameniable to well understood blackbox techniques.

In Section 3 we introduce a language that allows a family of equivalent programs to be defined using a non-deterministic choice of recursive function calls. The trace function defines a mapping from this family of programs onto a set of instruction sequences. Given an appropriate memory mapping (i.e. after register allocation), the "thin" nature of the VM means that the instruction sequences are easily transformed into executable programs for the target platform. Different evaluation orders of the trace function enumerate the set of solutions in different orders. The problem of searching the solution set is thus transformed into a backtracking search over the evaluation of the trace function. Our implementation in Prolog is a direct expression of this search.

The size of the search problem makes brute-force intractable. As this approach represents a novel method of constructing and testing candidate implementations it is currently an open question whether or not there exist efficient search algorithms. In this initial work we have taken a coarse sample of the search space for the example program. Because the program is the implementation of the simplest element in the problem domain we hypothosize that it contains the least potential for performance increase. Intuitively we claim that the pure solutions appear to be efficient, and there is no obvious scope for speedup by splicing them together.

Our prototype simply enumerates the first 500 instruction sequences found by a breadth-first evaluate of the trace function. By demonstrating that such a simplistic approach can produce convincing performance increases, where none are obvious to the human programmer, we hope to convince the reader that Program Interpolation is an area worthy of investigation. As this paper represent a (small) first step in the area, there are many unaswered questions. It is currently unclear exactly what the search complexity is for a given input program. It is unclear how to effectively perform that search. It is also unclear what the overall scope of the method is — how many useful programs can be written in this way.

It is interesting to compare this approach with the similar, brute-force approach of superoptimisation. In superoptimisation the search space is that of the set of all programs. Before a candidate solution can be evaluated, it must first be determined to be equivalent to the target program using a sequence of (probabilistic) tests. Massalin noted [14] that a more efficient approach would be to "bias" the generation of candidates to reduce the size of the search space. In a sense this is what our approach achieves: by deferring the definition of the bias to the programmer through provision of overlapping implementation strategies, enough information is presented to the compiler to efficiently realise the search.

Once a search strategy has been selected for the input problem it is necessary to measure each solution in order to generate a ranking. Our approach is to generate the program corrosponding to each instruction sequence, and then profile it on the target architecture. Using dynamic feedback in this manner allows us to expose complicated arhitectural artifacts without needing to construct a detailed model of the target processor. Our initial choice of measurement is execution time — other interesting candidates are power consumption and cache behaviour. For the maximum benefit the property chosen must be complex enough that the programmer cannot simply reason which instruction sequence is optimal.

In order to yield an accurate measure of performance, we use hardware based performance monitors such as the `rdtsc` instruction on IA-32 based processors [11]. Instrumenting an instruction sequence with such instructions allows measurement to a granularity of roughly 1000 clock-cycles. To improve the accuracy an average of many executions is used, we avoid the common approach of taking the mean execution time as the noise in measurement is not Guassian, but rather a skewed unbalanced distribution. To resolve this problem, we repeat the measurement over varying numbers of executions and use linear regession to determine the time to with a few percent of a clock-cycle.

5. Experimental Results

Our prototype interpolator was applied to the example program in Figure 1 and tested on two architecture with differing performance characteristics. Platform $\boldsymbol{A}$ housed a 2.80GHz Intel Pentium 4 processor, with a 32-bit install of Linux. Platform $\boldsymbol{B}$ housed a 2.4GHz Intel Core2 Duo running a 64-bit install of OS-X 10.5. In both cases GCC was used to generate 32-bit executables with the test sequences embedded. The rationale for the selection is that both architectures differ greatly in their memory / instruction scheduling characteristics.

HYPOTHESIS 1. *A solution will exceed pure strategy performance.*

Our choice of example program within the chosen domain is designed to show that even the simplest programs offer optimisation opportunities that a conventional approach would find difficult to exploit. By considering two separate pure strategies that solve the problem, we have implicitly provided enough information about the problem to automatically construct a large set candidate solutions that implement hybrid strategies. If a candidate representing one of the pure strategies achieves the highest performance, there

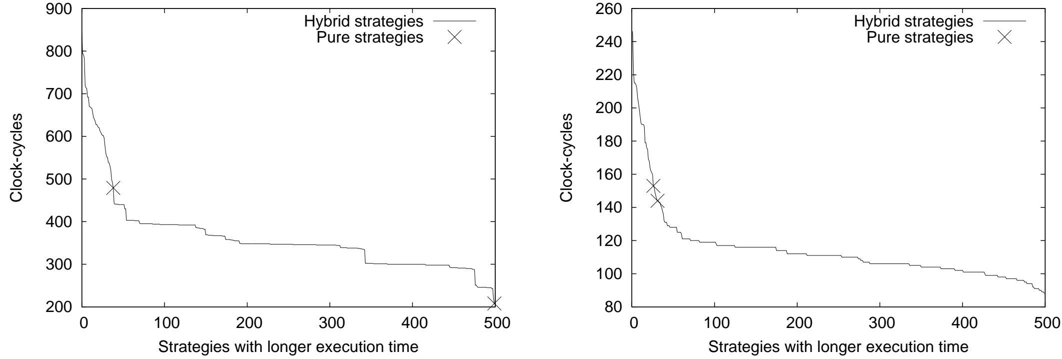

Figure 6. Graphs for Platform $\boldsymbol{A}$ (left) and Platform $\boldsymbol{B}$ (right) showing the ranking of solutions to the $m = 6$, $n = 10$ problem instance. For example, in the left graph 500 strategies take longer than 200 clock cycles, but only 1 strategy takes longer than 799 cycles.

was no benefit from interpolating the control-flows. On the other hand, if a candidate exists as a result of some hybrid strategy and that candidate achieves higher performance, exploration of the design space has clearly exposed some trade-off in the target platform. This would be a successful result in that it demonstrates sampling from a generated family of equivalent programs is a viable approach to program optimisation.

HYPOTHESIS 2. *The winning solution will exploit different trade-offs on the two platforms.*

Both platforms are superficially similar: both processors execute the binary-compatible IA-32 instruction set. Behaviourally the two platforms are completely different. Significant micro-architectural differences exist in the processor front-ends. For example Platform $\boldsymbol{A}$ includes a trace-cache and although Platform $\boldsymbol{B}$ does not, it includes other features to improve fetch and decode performance such as a loop buffer and improved use of instruction fusion. Instruction execution latencies are drastically different: the ratio of throughput between `ADD` and `ADC` is $10 : 1$ on the Platform $\boldsymbol{A}$ and $2 : 1$ on the Platform $\boldsymbol{B}$.

For our experiment we compile the example program with all parameterisations between $n = 4$ and $= 12$ digits, and between $m = 4$ and $m = 12$ variables. For each parametrisation, we generated a set of candidate solutions as outlined in Section 4, bounding the search to the first 500 candidates. Each candidate is measured and ranked according to execution time. Figure 6 details results for the specific $m = 6$, $n = 10$ case. The location of the pure strategies in the ranking is indicated by large $\times$ symbols. Each parametrisation produces a similar shaped distribution of performances with slight variations in each case. The absolute execution times are in different ranges and direct comparisons cannot be drawn between candidates on different platforms.

The result for Platform $\boldsymbol{A}$ shows that the digit-wise strategy was as fast as any of the generated hybrids; this implies that interpolation failed to beat the pure strategies for this platform. Further investigation showed that the large ratio between instruction latencies for `ADD` and `ADC` on this micro-architecture, combined with the memory sub-system performance, were significant in this failure. The result for Platform $\boldsymbol{B}$ supports Hypothesis 1 as interpolation generated many instruction sequences with higher performance than both pure strategies. In this case, the best sequence found took only 88 clock-cycles compared to 144 clock-cycles for the fastest pure strategy; this 39% decrease in execution time is a 63% increase in throughput for the selected problem.

This large performance increase is suprising for such a simple problem, but demonstrates the viability of our approach. Note the distribution for Platform $\boldsymbol{A}$ shows strange characteristics, in particular several sharp "performance thresholds" exist. We suspect that these correspond to bottlenecks in instruction issue and dispatch, and further examination of this distribution may reveal behavioural details about the platform. We stress that these details could not have been foreseen by the programmer; the automatic exploration of hybrid strategies has uncovered these features.

Figure 7 shows performance across a range of parameterisations. The number of variables is restricted to $m = 6$, but the number of digits varies from $n = 4$ to $n = 12$. For each case we plot the performance of the pure digit-wise strategy, the pure variable-wise strategy and the best candidate measured. This shows more clearly the interpolation failure on Platform $\boldsymbol{A}$: the best performing strategy is always the pure digit-wise strategy. The slice for Platform $\boldsymbol{B}$ shows that the ranking of strategies is dependent on the number of digits: once the number of digits exceeds $n = 4$, there exist hybrid strategies that perform better than the two pure strategies. The largest parametrisation in this set shows a 36% decrease in execution time for the best hybrid against a pure strategy; this is a 57% increase in throughput.

The final plots of the data-set are shown in Figure 8. For each platform we present three 2-dimensional surfaces plotted to show a comparison across all parametrisation. As with the single slice case above, the plot for Platform $\boldsymbol{A}$ shows that two surfaces are identical, i.e. the best solution is the pure digit-wise strategy. The best result in the entire data-set is the $m = 12$, $n = 12$ case where the best hybrid solution only requires 201 clock-cycles against the 327 clock-cycles required for the digit-wise case. This 39% decrease in time is a 62% increase in throughput.

6. Conclusions

A standard approach to optimisation split the problem into two parts: high-level optimisation is driven by humans who select appropriate algorithms, low-level optimisation attempts to automatically transform fixed implementations into more efficient forms. Modern hardware is complex enough that this division breaks down; high-level selection of an approach can be dominated by low-level performance characteristics. Rather than attempting to rewrite local instruction sequences in an undirected search we have proposed an alternative. This proposed approach is language-driven in that we allow the programmer to suggest multiple implementation techniques within a single program that communicate high-level optimisation decisions to the compiler so they can guide low-level optimisation therein. The experimental results from our prototype compiler show that impressive performance gains are possible even for a simple program.

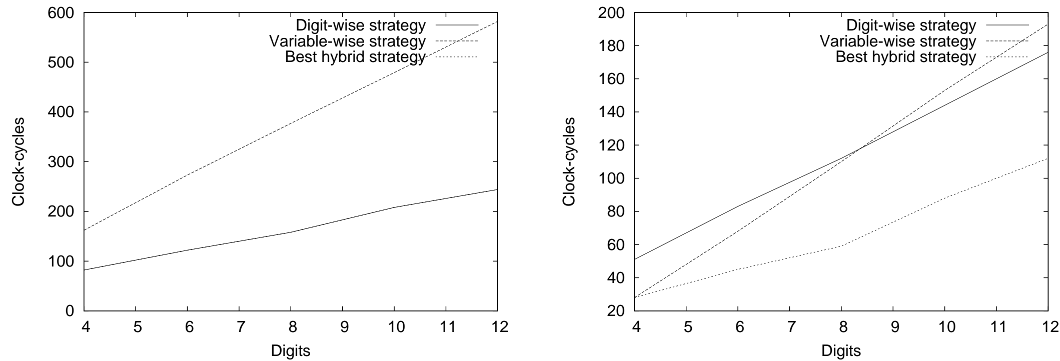

Figure 7. Graphs for Platform $\boldsymbol{A}$ (left) and Platform $\boldsymbol{B}$ (right) showing results for varying number of digits and $m = 12$ vectors.

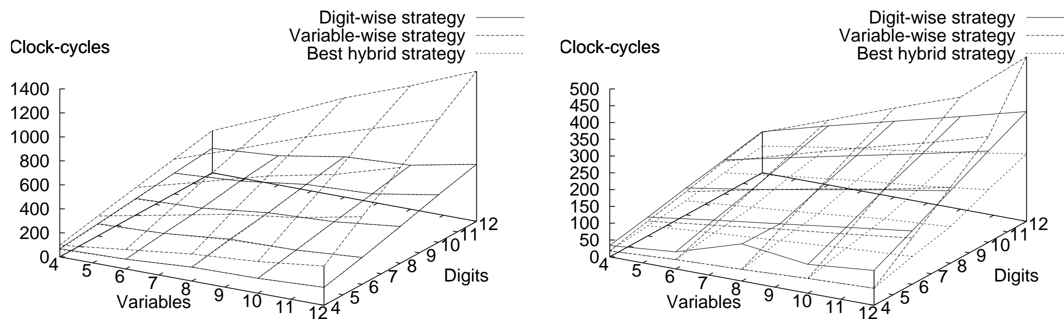

Figure 8. Graphs for Platform $\boldsymbol{A}$ (left) and Platform $\boldsymbol{B}$ (right) showing results for varying number of digits and vectors.

References

[1] D.J. Bernstein. Multidigit multiplication for mathematicians. Available at: `http://cr.yp.to/papers.html#m3`

[2] D.J. Bernstein. `qhasm`: tools to help write high-speed software. Available at: `http://cr.yp.to/qhasm.html`

[3] F. Bouchez, A. Darte, C. Guillon, and F. Rastello. Register Allocation and Spill Complexity Under SSA. Technical Report 2005–33, ENS-Lyon, Lyon France, 2005.

[4] P. Brisk, F. Dabiri, R. Jafari, and M. Sarrafzadeh. Optimal register sharing for high-level synthesis of SSA form programs. In *IEEE Trans. Computer Aided Design*, Vol. 25, no. 25, May, 2006, 772–779.

[5] P. Comba. Exponentiation cryptosystems on the IBM PC. In *IBM Systems Journal*, **29** (4), 526–538, 1990.

[6] Computational Algebra Group, University of Sydney. *Magma Computational Algebra System*. Available at: `http://magma.maths.usyd.edu.au/magma/`

[7] P. Gaudry and E. Thomé. The $\mathtt{mp}\mathbb{F}_q$ Library and Implementing Curve-based Key Exchanges. In *Software Performance Enhancement for Encryption and Decryption (SPEED)*, 49–64, 2007.

[8] N. Gura, A. Patel, A. Wander, H. Eberle, S. Chang Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems (CHES)*, Springer-Verlag LNCS 3156, 119–132, 2004.

[9] S. Hack and G. Goos. Optimal register allocation for ssa-form programs in polynomial time. In *Information Processing Letters*, 98(4):150-155, May 2006.

[10] D. Hankerson, A. Menezes and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.

[11] Intel Cooperation. Using the RDTSC Instruction for Performance Monitoring. Intel Technical Report, 1997.

[12] A. Karatsuba and Y. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. In *Doklady Akad. Nauk SSSR* **145**, 293–294, 1962.

[13] V. Lifschitz, D. Pearce and A. Valverde. Strongly equivalent logic programs. In *ACM Trans. on Computational Logic (TOCL)*, Vol 2, Issue 4 (Oct 2001), 526–541

[14] H. Massalin. Superoptimizer - A Look at the Smallest Program. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 122-126, 1987.

[15] A.J. Menezes, P.C. van Oorschot and S.A. Vanstone. *Handbook of Applied Cryptography*, CRC Press, 1997.

[16] A. Moss and H. Muller. Efficient Code Generation for a Domain Specific Language. In *Generative Programming and Component Engineering*, Springer-Verlag LNCS 3676, 47–62, 2005.

[17] S. Peyton Jones, N. Ramsey and F. Reig. C−−: A Portable Assembly Language that Supports Garbage Collection. In *Principles and Practice of Declarative Programming (PPDP)*, 1–28, 1999.

[18] R. Rivest, A. Shamir and L. M. Adleman A Method for Obtaining Digital Signatures and Public-key Cryptosystems. In *Communications of the ACM*, **21** (2), 120–126, 1978.

[19] M. Scott and P. Szczechowiak. Optimizing Multiprecision Multiplication for Public Key Cryptography. In *Cryptology ePrint Archive*, Report 2007/299, 2007.

[20] L. Uhsadel, A. Poschmann and C. Paar. Enabling Full-Size Public-Key Algorithms on 8-Bit Sensor Nodes. In *Security and Privacy in Ad-hoc and Sensor Networks (ESAS)*, Springer-Verlag LNCS 4572, 73–86, 2007.