SUPER COMPUTER

Contributions

Individual synthetic benchmarks and partial decompilation

Alexander Obé, Wolfgang Paul

Institute for Computer Architecture and Parallel Computers, Department of Computer Science, Universität des Saarlandes, PO Box 1150 D-66041 Saarbrücken 11, Germany

Supercomputer **56/57**, X–4
Received November 1993

In this paper we propose the following new approach for the construction of synthetic benchmarks: i) collect traces T on the system bus by monitoring hardware. ii) construct high level language programs P which generate the same workload as the traces T. iii) construct the synthetic benchmark from the programs P. Steps i) and iii) are straightforward. We present an algorithm for step ii) and report experimental results which are encouraging. This opens the way to the automatic construction of personal synthetic benchmarks.

Benchmarks are models of workload. In particular synthetic benchmarks are programmed statistics about the relative frequency of language constructs in certain programs. Computer architects optimize their machines in order to perform well on certain benchmarks. Buyers frequently base their decisions on benchmark performance, although it is not clear how well their own workload is represented by the benchmarks [1–3].

A buyer who knows enough about the behaviour of his programs can easily construct and run a synthetic benchmark which represents his own workload. Gathering such statistics is unfortunately not a trivial matter. From hardware monitors one can immediately obtain statistics about the machine instructions executed, but this is of little value if one wants to test a machine with a different instruction set. Instrumenting source code yields the desired data, but this approach may be impractical for a variety of reasons: source code may not be accessible, or the penalty in run time may not be acceptable.

In this paper we present a method for transforming a trace T, obtained by a hardware monitor, into a high level language program P which produces the same workload as trace T. This is done by a process we call partial decompilation. It is similar to decompilation [4, 5], but, however, easier because semantics do not have to be preserved.

We will only describe partial decompilation from a particular RISC instruction set to C. In the next section we justify this. Then we describe the method and heuristics we have used. Finally we report experimental data. In section 10 we discuss limitations of this approach and possible future work.

This work was partially supported by BMFT project PARANUSS.

Programming languages and machine languages

Computer programming is clearly dominated by imperative programming languages. The object oriented programming languages can be considered as extensions to imperative programming languages. C, Fortran, and Pascal are typical representatives for imperative programming languages and are based on common constructs as demonstrated in Figure 1.

Property	C	Fortrn	Pascal
predefined data types	(unsigned) char, (unsigned) int, (unsigned) long, float, double	char, integer∗1/2/4, real∗4/8, complex, logical	char, integer, real, boolean
userdefined data types	vector, struct	vector (dim)	array, record
arithmetic–logical operators	$+, -, *, /, \%,$ $\ll, \gg, <, >, \leq, \geq,$ $==, !=, \&, \hat{}, \mid,$ $!, \sim, -(unary)$	$+, -, *, /,$ lt, gt, le, ge, eq, ne, and, xor, or, not, $-(unary)$	$+, -, *, /,$ div, mod, $<, >, \leq, \geq, =, <>,$ and, or, not, $-(unary)$
iteration	for, while, while – do	do ... label	while, repeat – until for
branching	if – else, switch – case, goto	if – then – else, goto	if – then – else, case, goto
functions, procedures	function	function, subroutine	function, procedure

Figure 1. Common constructs of imperative programming languages.

The algorithm for partial decompilation presented later, will essentially assume that the trace T is generated by a compiled program from any imperative language. The algorithm will try to construct a C program which produces the same workload. Thus it has heuristics for recognizing the kinds of constructs from the first column in Figure 1. If all heuristics fail, C permits to closely mimic assembler instructions as a last resort. One construct the present algorithm fails to treat correctly is the visibility of variables in Pascal. We will return to this point in section 10.

Most of the commercially successfull processor architectures mirror the common constructs of imperative programming languages. As a selection of those processors we consider the set $P = \{$MOTOROLA 68040, SPARC, MIPS, INTEL 80486, INTEL i860$\}$. In [6] the authors observe that the RISC processors SPARC, MIPS, i860, and M88000 all have similar instruction sets. They proceed to define an instruction set DLX which captures the common features of these instruction sets.

In this paper we assume that the instructions in trace T are DLX instructions. In DLX memory is addressable by bytes. All instructions are 4 bytes long. Register 0 is always 0. There is a special set of floating-point registers and there are special registers for status (sr) and the stack pointer (sp). There are four classes of instructions, namely i) loads, ii)

stores, iii) arithmetic/logical instructions and iv) branches. We call instructions of types i) to iii) computational. Loads and stores move 1, 2, 4 or 8 bytes between CPU registers and memory locations. There is a single addressing mode: register + constant. There is no load immediate instruction. Arithmetic/logical operations have 2 operands. They are between registers only, but they may have one immediate operand (thus load immediates can be simulated). They include compare instructions which change the content of the status register. Conditional branches depend on the content of the status register. Procedure calls are supported by special jump and link instructions.
The problem to partially decompile traces T coming from a complex instruction set is easily reduced to the problem to decompile from DLX: replace each instruction in T by a short equivalent sequence of DLX instructions, then partially decompile the resulting sequence T'.
Our experiments are based on the SPARC instruction set.

Code generation, object code and traces

DLX and similar instruction sets allow the straightforward translation of high level language constructs:

- Simple and complex data objects of high level languages are mapped to memory locations.
- Mapping data to memory locations requires additional operations for memory management and addressing operations. The instructions SLL, SRL, ADD, and the data transfer instructions meet that requirement.
- Simple logic and arithmetic operations of high level languages correspond to the identical DLX instructions.
- High level language branches and loops may be realized using the (conditional) jump instructions of DLX.
- Function calls are carried out using jump and link instructions.

Earlier studies on decompilers ([4, 5]) always refer to object code. A trace drawn from a running program differs from object code because some parts of the program may never be executed while others may be executed repeatedly. For each machine instruction monitored we assume that the following triple $t = (CA(t), C(t), DA(t))$ is recorded: the code address $CA(t)$ from which the instruction is fetched, the code $C(t)$ executed (a DLX instruction) and the (possibly empty) data address $DA(t)$ accessed in case $C(t)$ is a load or store operation. A trace T then is simply a sequence of such triples t. Different triples with identical code address come from repeated execution of the same instruction I in the object code. For triples t and t' we write $t < t'$ if t occurs befor t' in the trace. For each code address b we denote by $C(b)$ the instruction stored at address b.

System load and user load

In our experimental work we have to take into account that a workload is separated into system load and user load. System load consists of

the operating system processes. Given a Unix system, the system load is easily distinguished from the user load. Unix systems strictly separate user and system address space. Furthermore Unix defines a clear interface between user and system processes: system calls. Looking at the DLX instruction set, system calls may be carried out by the TRAP instruction, the RFE instruction returns to the user process. Thus the fraction of the trace representing the system load may easily be recognized and isolated. User load is then the remaining part of the trace. In the next 4 sections we present a method for decompiling the portion of the trace which corresponds to user load.

Program graphs and loops

From trace T we generate a program graph $G = (V, E)$. The nodes V of G are the different code addresses occuring in T. We include an edge from a to a' if at some place in trace T a triple with code address a' immediately follows a triple with code address a. Such an edge also indicates that in the object code one can branch from a to a'. For real machines which pipeline the execution of instructions, this definition is more complicated due to delay slots.

Strongly connected components of graph G correspond almost, but not exactly to the outermost loops of the object code respectively the source code. Suppose the same function is called from code addresses b and c. Then we have in G a directed path from b to $c + 4$, i.e. to the return address belonging to the call from code address c. This is easily fixed: for each node a such that $C(a)$ is a jump and link instruction delete all edges which start in a and add the edge $(a, a + 4)$. If E' is the new set of edges, then outermost loops correspond directly to strongly connected components of the graph (V, E').

Breaking the outermost loops of strongly connected components in appropriate places and searching for strongly connected components in the remaining components permits to identify nested loops.

Basic data structures

The next three sections describe the central portion of our algorithm. The algorithm processes the triples t in the order of their occurrence in trace T. While doing so it maintains several data structures:

Functions. In order to keep track of the nesting of function calls and of return addresses we maintain two stacks and two numbers:

- A nesting depth s of function calls. Initially $s = 0$.
- The number n of function calls observerd so far. Initially $n = 0$.
- A "function" stack F and
- A stack R of return addresses.

Whenever we process a triple t such that $C(t)$ is a jump and link instruction we have identified the call of a function. This function has nesting depth $s + 1$, its body starts at address $CA(t + 1)$ and after return from this function the instruction stored at code address $CA(t) + 4$ will be

executed. (Again, we ignore delay slots.) We increment s and n. Then we push the triple $q = (CA(t+1), n, s)$ on the function stack. We push $CA(t) + 4$ on the stack R.

Whenever we process a triple t such that $CA(t) = top(R)$, we have identified the return from a function. We pop F and R and we decrement s (but not n).

If $q = (CA(t+1), n, s)$ is a triple on the function stack, then $CA(t+1)$ specifies the (start address of the) function, n specifies the instance of that function and s specifies the nesting depth of the currently observed call. We order the entries of the function stack F by their third components, i.e. smaller entries have smaller nesting depth. For entries $q = (f, n, s)$ of the function stack we use the notation $f = f(q)$, $n = n(q)$ and $s = s(q)$.

Last changes of locations. We maintain a pair of mappings $lastca()$ and $lastf()$. Whenever a triple $t = (CA(t), C(t), DA(t))$ modifies a register or a memory location l then $lastca(l)$ is updated to $CA(t)$ and $lastf(l)$ is updated to $top(F)$. If l is a memory location, then $C(t)$ is necessarily a store instruction, otherwise $C(t)$ is necessarily a load or an arithmetic/logical instruction.

Straight line segments. Let b be a code address, respectively a node of the computation graph. We call b computational if $C(b)$ is a computational instruction. A straight line segment is a sequence $(a_1, ..., a_{max})$ of code addresses such that

1 a_i is a computational instruction for all i;
2 neither $a_1 - 4$ nor $a_{max} + 4$ are computational;
3 $a_i = a_{i-1} + 4$ for all $i > 0$.

We partition the computational nodes into straight line segments in the order of their occurrence in the trace. Note that straight line segments are not necessarily basic blocks, because interior nodes of straight line segments can be destinations of branch instructions. It is easy to maintain the start end end points of the straight line segments already found in a dictionary like data structure.

With the help of this data structure and function $lastca()$ one computes a function $lasts(l) =$ the straight line segment which contains code address $lastca(l)$, i.e. the straight line segment in which location l was modified last.

In the central loop of our algorithm we will process straight line segments in the order in which they appear in the trace. When a straight line segment appears a second time, we only update the above data structures.

Identifiers and expressions. Let b be a code address respectively a node in the computation graph. We associate with each such address b and with each straight line segment A an identifier $I(A, b)$. Some of these identifiers will later be turned into simple variables or into components of complex variables.

If code address b belongs to straight line segment A, then $I(A, b)$ serves simply as a temporary variable which holds the value computed by in-

struction $C(b)$ during execution of straight line segment A. But there are four occasions, where we use during execution of straight line segment A a value $I(A', b')$ which was computed by instruction $C(b')$ during execution of straight line segment A'. Let $q = (f, n, s) = TOP(F)$ at the time we process A and let $q' = (f', n', s') = TOP(F)$ at the time we process A':

1 Segment A accesses a global variable which was computed by segment A'.
2 function f' called function f and segment A accesses an actual parameter computed by function f'. In this case $I(A', b')$ will eventually become a formal parameter of f.
3 function f called function f' and $I(A', b')$ is the value returned from f' to f. In this case $I(A', b')$ will eventually become the name of the called function f'.
4 $n = n'$. Straight line segment A accesses a value computed by a previous segment A' during the actual function instance.

We will compute for each code address b and some segments A a triple $(e(A, b)$, type(A, b), class$(A, b))$ where

1 $e(A, b)$ is an expression from the following set Ex: all constants and all identifiers $I(A, b)$ are in Ex. If $e1$ and $e2$ are in Ex and op is an arithmetic/logical operation of DLX, then $op(e1, e2)$ is in Ex. The expressions $e(A, b)$ serve four purposes:
   – to construct right hand sides of assignment statements (such a statement is generated for every store instruction);
   – to construct conditions in `if`-statements and loops;
   – to form actual parameters in function calls;
   – (in case instruction C(b) belongs to an address computation) to determine storage classes of variables.
2 type$(A, b) \in$ {*char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, double*} approximates the type of $I(A, b)$.
3 class$(A, b) \in$ {*constant, local, global, register, local array, global array, by local pointer, by global pointer*} approximates the storage class of $I(A, b)$ or of the complex variable of which $I(A, b)$ is a component. We use the notation $Gl =$ {*global, global array, by global pointer*}.

The central loop

For each triple t of the trace, such that $C(t)$ is a computational instruction belonging to a straight line segment A which was not processed before, we simultaneously do three additional things:

1 We construct for A a computation dag G(A) in a fairly straight forward way.
2 We inspect the sources of this dag and identify global variables, parameters and returned values of functions.
3 We construct $e(A, b)$, type(A, b) and class(A, b) for the nodes $b \in A$.

Construction of computation dags $G(A)$. We assume that triple t is being processed and that a (possibly empty) initial segment of A has already been processed. Let V respectively E be the set of nodes respectively edges of the dag $G(A)$ constructed so far (initially $V = E =$ empty).
The code address $CA(t)$ is included into V.
If $C(t)$ is a load instruction and $c = lastca(DA(t))$ is defined, then c is included in V (it may or may not already be in V) and the edge $(c, C(t))$ is included into E.
If $C(t)$ is a store instruction, which writes into memory from register r, then $c = lastca(r)$ is included into V and edge $(c, C(t))$ is included into E. If we have the trace of a complete run of a program, then c usually is defined.
If $C(t)$ is a computational instruction which applies an operator op to registers r and r' then $c = lastca(r)$ and $c' = lastca(r')$ are included into V and the edges $(c, C(t))$ and $(c', C(t))$ are included into E. If the second operand is an immediate value, then only c and $(c, C(t))$ are included into V respectively E.

Identification of parameters and returned values of functions
Whenever we include into V a node $c = lastca(DA(t))$ or $c = lastca(r)$, which does not belong to the current straight line segment, we are accessing a value, which was computed earlier by the instruction stored at code address c. Essentially we want to determine if in the decompiled program we want to introduce a new identifier for the value stored at location c (formally $e(A, c) = I(A, c)$) or if we want to reuse the old identifier (formally $e(A, c) = I(A', c)$, where A' is defined below)).
Let $l = DA(t)$ respectively $l = r$ be the location which is accessed, let $q' = (f', n', s') = lastf(l)$ be $TOP(F)$ at the time when that location l was modified last and let $A' = lasts(l)$ be the straight line segment in which l was modified last. Then A' belongs to function f' and $e(A', l)$, type(A', l) and class(A', l) are all defined because of the order in which we process straight line segments. We abbreviate with $q = (f, n, s)$ the current $top(F)$. There are several cases:

1 If class$(A', c) \in Gl$ then $I(A', c)$ is a global variable. We do not use a new variable and set $e(A, c) = I(A', c)$, type(A, c) = type(A', c) and class(A, c) = class(A', c).
2 If class$(A', c) \notin Gl$ and $s' < s$, then the current value of l was computed in a function instance of smaller nesting depth. The current content of location l was put on the stack or into register r during segment A' and is now an actual parameter of the current function instance Q. We create a local parameter $I(A, c)$. In the corresponding call of function f in the body of f' the actual parameter is $I(A', c)$ if l is a memory location and $e(A', c)$ if l is a register. We set $e(A, c) = I(A, c)$, type(A, c) = type(A', c) and class(A, c) = local.
3 If class$(A', c) \notin Gl$ and $s' > s$ then the current value of l was computed in a function instance of larger nesting depth. Hence it is the value returned by f' to the current function f. We create a

synonym $I(A,c)$ for the name of the function f' (i.e. eventually in the decompiled program $I(A,c)$ is replaced by a call of function f'). In the body of function f' we use in the return statement $I(A',c)$ if l is a memory location and expression $e(A',c)$ if l is a register. We set $e(A,c) = I(A,c)$, type(A,c) = type(A',c) and class(A,c) = class(A',c).

4 If $n = n'$ then the current content of location l was computed in the current function instance. We do not use a new identifier $I(A,c)$. We set type(A,c) = type(A',c) and class(A,c) = class(A',c). If l is a memory location we set $e(A,c) = I(A',c)$, if l is a register we set $e(A,c) = e(A',c)$.

Computation of expressions, classes and types for nodes in the current straight line segment. Let b = CA(t). There are 3 cases:

1 Loads. If there is no edge (c,b) in E, then $lastca(DA(t))$ is not defined and we have found a new variable or a component of a complex variable. If we have the trace of a complete run of some program then triple t accesses input data. Formally we set $e(A,b) = I(A,b)$ and we determine type(A,b) and class(A,b) with the heuristics from the next section.
If there is an edge (c,b) we simply set $e(A,b) = e(A,c)$, type(A,b) = type(A,c) and class(A,b) = register.

2 Arithmetic/logical instructions. Suppose instruction $C(t)$ computes $op(r,s)$ and writes the result into register l. If and r and s are both registers we set $e(A,b) = op(e(A,r),e(A,s))$ and we determine type(A,b) by the heuristics of the next section.
A few special cases arise from the way in which DLX simulates load immediate instructions. If s is constant we set $e(A,b) = op(e(A,r),s)$. If moreover $e(A,r)$ is constant, then we infer type(A,b) = *double* if l is a floating-point register with an even number, *float* if l is a floating-point register with an odd number and *int* otherwise. If s is an immediate value and $e(A,r)$ is not constant we temporarily infer for s type *int* and then we infer type(A,b) with the heuristics of the next section.
In all cases we set class(A,b) = register.

3 Stores. If instruction $C(t)$ stores the content of register r into memory location $l = DA(t)$ several cases arise:
- $lastca(l)$ is not defined or $lastf(l) \neq$ the actual $TOP(F)$. Then we use the new identifier $I(A,b)$, we set $e(A,b) = I(A,b)$, type(A,b) = type(A,r) and we determine the class(A,b) by the heuristics of section 8. In the decompiled program we generate the assignment statement $I(A,b) = e(A,r)$.
- $c = lastca(l)$ is defined and $lastf(l) =$ the actual $TOP(F)$. Then there was a write to this memory location in the actual function instance. Let $A' = lasts(l) =$ the segment where this write occured. Then we reuse the identifier $I(A',c)$. We set $e(A,b) = I(A',c)$. In the decompiled program we generate the assignment

statement $I(A', c) = e(A, r)$. *We do however update the type* of the reused variable $I(A', c)$ with the type of the present content of r: let $c" = lastca(r)$ be the address of the instruction which changed register r last and let $A" = lasts(r)$. Then we update type(A', c) = type$(A", c")$. In this way we propagate type information gained during the processing of computational instructions back to (some of) the variables.

Dedicated registers get the obvios special treatment: $e(A, Register0) = 0, e(A, sp) = sp, e(A, sr) = sr$ for all segments A.

In our experimental work we have used a radically simplified version of this algorithm in order to save space and computation time. We jumped over all straight line segments processed before and did not update functions $lastca()$ and $lastcf()$ in the process. One can construct examples where this produces undesired results. Nevertheless even this procedure gave very satisfactory results (see the section on experimental results).

Two further complications arise if one considers traces which do not come from complete runs of a program:

1 The nesting depth s can become negative. The easy way out here is to process the the trace only up to this point.
2 Now very often $lastca(l)$ and $lastf(l)$ are undefined although there were writes to location l before the trace was taken. In particular there are reads from registers r without previous writes. In this situation one has to use variables not only for memory locations but also for registers. In [7] these variables are called pseudovariables.

Heuristics for type and class of variables

We first review how code generated by compilers accesses variables with various types and storage classes. Then we point out how we generate expressions $e'()$ which describe address computations. Finally we state the heuristics we use.

Memory segmentation and memory access. A compiler usually maps a contiguous address space into four segments:

1 The code segment holding a program's instructions. The code segment is accessed via the program counter *pc*.
2 The data segment holding global data, i.e. data accessible from all functions of a program. Variables in the data segment are either accessed using a constant address or using a *basepointer/offset* combination.
3 The stack segment holding local data for every active function call. Local variables are usually addressed via a *stackpointer/offset* combination.
4 The heap segment holding variables that are dynamically allocated during the runtime of a program. Since heap variable addresses are not known when compiling a program, they must be accessed using some intermediate variables: pointer variables. The addresses of heap variables are assigned to pointer variables when those addresses are determined by a memory allocating function.

Local variables may also be addressed using a *stackpointer/offset* combination. Since *framepointer* and *stackpointer* are basically identical – the calling function's *stackpointer* is often the called function's *framepointer* – we do not distinguish *framepointer* and *stackpointer*.

Arrays and structs.
Arrays and structs are collections of variables. All components of an array have the same type. Structs have typically (but not always) components of different types. Structs and arrays are accessed in a similar way in memory. An address specifying the collections start address is used as a base. Components of the collection are accessed by adding an offset to the base. We call a variable a *complex variable* if it is accessed using a base. We call it *simple* if it is not complex.
Recall that we use storage classes from the set {*constant, local, global, register, local array, global array, by local pointer, by global pointer*}. *Local* variables are simple variables on the stack, *global* variables are simple variables in the data segment. *Local array* specifies complex variables in the stack segment, *global array* complex variables in the data segment. *Local pointers* specify complex variables in the heap segment, that are accessed via a local base. *Global pointers* access complex heap variables accessed via a global base. We do not distinguish between arrays and structs yet.

Expressions for address calculations. In order to infer the storage class of variables we have to consider the address calculations which are performed by the addressing mode register + constant of DLX. While a triple t is processed in the central loop the obvious extra nodes and edges which come from these address calculations have to be included into the dag $G(A)$:
if a load or store instruction $b = C(t)$ uses address computation $r + k$ where r is a register and k is a constant we include the node $c = lastca(r)$ into V and the edge (c, b) into a new set E' of edges which come from address calculations. For c we determine $I(A, b), e(A, b)$ etc. as above. The address used by instruction $C(t)$ is desribed by $e'(b) = e(A, c) + k$.

Determining the variable type. The type of data loaded from memory is inferred from the type of the load operation:

Load operation	**Variable type**
load byte (unsigned)	(unsigned) char
load halfword (unsigned)	(unsigned) short
load word	int
load float	float
load double	double

In our experimental work which is based on the SPARC instruction set, we infer for load immediate instructions the type *int* if ≤ 32 bits were loaded and *double* otherwise.
We infer the type t of the result of an arithmetic logical operation op in

the following way:
If op is a single (double) precision floating-point operation then t =float (double). If op is not a floating-point operation and both operands have the same inferred typt t', we infer $t = t'$.
If the operands have different inferred types $t1$, $t2$ we have to resolve the inconsistency. If one of the operands is constant and the other one is not, we infer $t =$ the type of the non constant operand, i.e. we do not place high confidence in the inferred type of constants. Otherwise we infer $t =$ the *shorter* one of the types $t1$, $t2$.
This heuristic for resolving inconsistencies performed in our experiments better than the opposite pattern, i.e. the longer type dominates the shorter one. A reason for that may be the 32-bit architecture of the processors $\in P$. A compiler will – for efficiency reasons – try to use a *typical* word length for the given architecture. According to the C language this is the type *int*. On the other side a programmer may use short and long types. Since we aim at generating C constructs close to the original code underlying the trace, we choose the variable type that conforms more to the programmer than to the architecture.
If an inconsistency was resolved, the effect is propagated to some extent backward by the mechanism described in the previous section (stores), for example if $X = XopZ$ was computed and we inferred the type of X incorrectly during the load of X. Clearly, we place higher confidence in types inferred from loads *and* arithmetic/logical operations than in types inferred from loads alone.

Determining the storage class. In order to determine storage classes, we analyze the addressing computations. The storage classes *local, global, register* are easily recognized:

- *local* variables are accessed using $[fp + const]$ or $[sp + const]$.
- *global* variables are accessed using [const] or $[bp + const]$.

All variables that do not fit in the given pattern must therefore be *local array, global array, by local pointer* or *by global pointer*. The distinction in *local* and *global* depends on the base address. The distinction in *pointer* and *array* depends on the variable being located in the heap or the stack segment respectively.

storage class	*local array*	*by local pointer*	*global array*	*by global pointer*
base address	$sp + const$	$[sp + const]$	$const/bp + const$	$[const]/[bp + const]$

We do not try to determine storage classes more precisely, i.e. to distinguish matrices, vectors, and structs. The reason is that there is not a unique addressing of elements of those structures. For a matrix for example one can easily think of two or three ways of addressing a matrix element.
We try to distinguish base and offset in an addressing expression with the following heuristic: that variable is a base that is not shifted left by a power of two. Leftshift is the typical operation for computing an offset. An index has to be adjusted to the accessed variable type. Since the

word length of every variable type is a power of two, leftshift is used to do the adjustment. This heuristic allows to isolate bases for all complex variables that do not access a variable of type *char*. It also works for *packed* strings of characters.
Now we also can identify pointers. They are simply variables which are used as a base address.
Data address leveling. While we construct the program graph, we collect lists of data addresses $DAL(b)$ for every code address b such that $C(b)$ is a LOAD or STORE instruction. If such an instruction $C(b)$ has been executed several times, then list $DAL(b)$ contains all data addresses accessed by that single instruction.
So far only single variables and components of complex variables have been considered. We now collect all variables that are accessed using the same base into complex variables.
The resulting complex variables are tested or common subsets of their corresponding address lists. That way complex variables which are accessed in different instructions by different bases are identified. The complex variables are then classified as scalars, vectors, and structs.

1 A scalar is accessed if the complete data address list contains the same data address at all positions.
2 A vector is accessed if there is a constant difference between two adjacent address list elements. Since a vector may be accessed several times, the difference between two adjacent list element has not always to be constant.
3 A struct is recognized when variables of the different variable types are addressed using the same base.

If the above heuristics for determining a variable's storage class fail, we decompile (in desperation) by translating the address computation explicitely into C.

Experimental results

In order to verify the method for decompiling, traces of different programs have been generated on a machine of the SPARC architecture. The traces have been partially decompiled and the result has been compared with the original program.
The test included different loops of the Livermore Kernels ([8]) written in Fortran and C and compiled using different levels of optimization. Furthermore a trace of the Dhrystone benchmark ([9]) translated using a C compiler has been partially decompiled.
Decompiling a trace of e.g. *kernel 1* of the Livermore Loops – an inner product of two vectors – the loop controlling the product generation has been found exactly. All variables used and all operators were generated correctly. Vectors could be distinguished from scalars.
When partially decompiling a trace of the same loop written in Fortran and compiled with the highest level of optimization the following optimization were found:

– The loop was unrolled.

- The unrolled loop was executed at half the repetitions (as might be expected, when loop unrolling is used.)
- Scalars were held only in registers.

When partially decompiling a matrix multiplication (*kernel 21* of the Livermore loops) the decompilation result still corresponded very closely to the original, i.e. the matrices accessed were identified by their bases, the three loops controlling the matrix multiplication were correctly identified. The Dhrystone benchmark consists of a set of 14 functions calling each other. All function calls have been correctly identified. Only functions that are usually included by shared libraries (`strcmp`, `strcpy` e.g.) were hard to recognize, because of the administration overhead of a shared library system. Most variables of Dhrystone were correctly identified. Only matrix variables were not correctly found because only two elements of that matrix were accessed and therefore a meaningful address list was lacking.

Conclusion and further work

The algorithm which we have presented uses a straightforward approach to the problem of inverting the function of a compiler. At the technical level however the algorithm is involved and one has to get many details and heuristics simultaneously correct. We have little hope to simplify the algorithm substantially. After all it must handle at least as many different cases as there are constructs in imperative programming languages.

In the few cases where we tested our algorithm, it even preserved semantics. In general we do not expect this to be the case, but we expect workload to be preserved fairly well. In our examples all statistics on which the Dhrystone benchmark is based were preserved as long as no highly optimizing compiler was used. Unnecessary work eliminated by a highly optimizing compiler can of course not be reintroduced by the decompiler.

We can presently not present more experimental data because the result of the partial decompilation as described here (and as implemented so far) is only a *fragment* of a correct C-program. It does for instance neither have declarations nor initial values of variables. Thus the result of the partial decompilation has presently to be inspected by a tedious manual process. We intend to produce a postprocessor in the spirit of [10] which produces complete decompiled C-programs and then to run more extensive tests.

So far our results suggest, that personal synthetic benchmarks can be constructed automatically from data monitored on the CPU bus. Such benchmarks at least have a known relation to the own workload as opposed to synthetic benchmarks based on the workload of others. There are four limitations to this approach, one of them inherent:

1 Work eliminated by an optimizing compiler can in general not be recovered (there is for instance no way to reconstruct eliminated dead code).

2 Certain constructs from Pascal do not occur in C. If a compiled Pascal program follows pointers in order to find a global parameter then the

present decompiler will fail to recognize this, and one can construct cases, where wrong workload is produced. But one is not forced to decompile to a single language only.

3 Decompiling cannot be done in real time and completely tracing big runs of programs is completely impractical. If workload stays reasonably uniform over time, then this can be overcome by tracing and decompiling parts of runs at random times.

4 On new processor chips with on chip cache one cannot monitor the CPU bus any more. This difficulty can at least in priciple be overcome by the manufacturer of the chip. Also for processor architectures which are around for a long time (e.g. SPARC) one can easily find machines, whose processors do not have on chip cache. On these machines one can collect the traces.

1
Ferrari, D., *On the foundations of artificial workload design*, in: "Proceeding of the 1984 SIGMETRICS Conference", 8–14, ACM SIGMETRICS, 1984.

2
McDonell, K.J., *Taking performance evaluation out of the 'stone' age*, in: "Proceedings of the summer 1987 USENIX Conference", 407–417, USENIX Association, June 1987.

3
Erhard, W., A. Grefe, M. Gutzmann and D. Pöschl, *Vergleich von Leistungsbewertungsverfahren für unkonventionelle Rechner*, Arbeitsberichte des Instituts für mathematische Maschinen und Datenverarbeitung (Informatik), **25 (5)**, Institut für Mathematische Maschinen und Datenverarbeitung, Erlangen, Juli 1991.

4
Hollander, C.R., *Decompilation of object programs*, PhD thesis, Stanford University, January 1973.

5
Claus, Lichtblau, and Wankmüller, *I. Bericht zur Aufwärtsübersetzung*, Technical Report **119**, Universität Dortmund, Abt. Informatik, 1981.

6
Hennessy, J.L. and D.A. Patterson, *Computer Architecture, a Quantitative Approach*, Morgan Kaufman Publishers, Inc. 1990.

7
Obé, A., *Characterisierung von Arbeitslast durch partielles Decompilieren zur Leistungsbewertung von Rechnern*, PhD thesis, Universität des Saarlandes, Saarbrücken, 1992.

8
McMahon, F.H., *The Livermore FORTRAN kernels test of the numerical performance range*, in: Martin J.L. (ed.), "Performance Evaluation of Supercomputers", 143–186, Elsevier Science Publishers, 1988.

9
Weicker, R.P., *Dhrystone: A synthetic systems programming benchmark*, Communications of the ACM, **27(10)** 1013–1030, October 1984.

10
Baker, B.S., *An algorithm for structuring flowgraphs*, J. ACM **24(1)**, 98–120, 1977.