Optimising Hot Paths in a Dynamic Binary Translator[*]

David Ung and Cristina Cifuentes[†]

Department of Computer Science and Electrical Engineering

University of Queensland, QLD, Australia

{davidu, cristina}@csee.uq.edu.au

ABSTRACT

In dynamic binary translation, code is translated "on the fly" at run-time, while the user perceives ordinary execution of the program on the target machine. Code fragments that are frequently executed follow the same sequence of flow control over a period of time. These fragments form a hot path and are optimised to improve the overall performance of the program.

Multiple hot paths may also exist in programs. A program may choose to execute in one hot path for some time, but later switch to another hot path, or even cycle between hot paths. Hence, each hot path is able to capture the most frequent execution pattern for a particular run-time stage of the program. The end result is that scattered code is collected, merged and optimised in a special way.

In this paper we describe the optimisations performed by UQDBT - a machine-adaptable dynamic binary translator. We provide an algorithm for finding hot paths using edge weight profiles, and we explain how to optimise code in a machine-independent way, based on hot path information.

Keywords

Dynamic compilation, run-time profiling, dynamic execution, binary translation.

1. INTRODUCTION

In dynamic binary translation, code is translated "on the fly" at run-time, while the user sees what appears to be ordinary execution of the program on the target machine. The dynamic translation process is performed in two stages: generating native code (translation) and executing it. Translations are performed in an on-demand basis during run-time. A dynamic translator may perform on-demand optimisations during code execution. The choice of optimisations that can be performed in a dynamic system is limited by the amount of time it can spend without the user noticing a drop in speed. As the majority of computer programs spend most of their time executing a small piece of code, optimising such code can substantially improve the overall performance of programs.

Code fragments that are identified to be frequently executed ("hot") follow the same sequence of flow control over a period of time. When a block of code becomes hot, some other blocks surrounding this hot block will also be hot. The sequence of executions through these hot code blocks follows a repeating pattern. This sequence of blocks forms a hot path – the unit for run-time optimisation. Once a hot path is identified, the code within it can be customised especially to reflect the flow path taken by that sequence. The end result is that scattered code is collected, merged and optimised in a special way.

Multiple hot paths may also exist in programs. A program may choose to execute in one hot path for some time, but later switch to another hot path, or even cycle between hot paths. Hence, each hot path is able to capture the most frequent execution pattern for a particular run-time stage of the program. As a program's behaviour changes, a new hot path is typically needed.

In this paper we describe the optimisations performed by UQDBT [20] - a *machine-adaptable* dynamic binary translator based on the static UQBT [14,19] framework. UQDBT supports different source and target machines through the specification of properties of these machines and their instruction sets. Translators such as FX!32[4] are specifically designed for two architectures and are closely bound to their underlying machines. Binary optimisers like Dynamo [21] and Wiggins/Redstone [22] are also written specifically for a fixed machine. Hence, their optimisations are designed and tailored to a particular set of hardware. In the case of UQDBT, the optimisations performed are generic and can be applied to

[*] This work was funded by an Australian Postgraduate Award and Sun Microsystems Inc.

[†] Now at Sun Microsystems Inc. cristina.cifuentes@eng.sun.com

other machines. Like Dynamo and Wiggins/Redstone, UQDBT concentrates the optimisation on sections of code that are found to be hot. UQDBT can also examine different stages of program execution and associates hot paths with each new stage.

The identification of hot paths in UQDBT is through the use of edge weight profiling, while other systems like Wiggins/Redstone use hardware counters (sampling by frequent interrupts to identify frequent values of program counter). The main advantage of edge profiles is that it provides more precise and complete information about hot paths than the sampling method. The disadvantage is that edge profiling is more expensive to compute since it keeps a complete history of all edge weights before the optimisation.

This paper is structured in the following way. Section 2 discusses the UQDBT framework for dynamic binary translation and its support for machine adaptability. Section 3 shows the algorithm for finding hot paths based on edge weight profiles. Section 4 describes how optimisations are implemented in UQDBT. Section 5 discusses some of the work related to the area of binary translation. Conclusions are given in Section 6.

2. FRAMEWORKS

Binary translators are like compilers, and like compilers, there are static and dynamic binary translators. In static binary translation, the code is translated off-line. The program to be translated is the source code to the compiler and the result of the translation is the output from the compiler. The translator converts the source program to create a new program (called a translation) that uses the machine instructions of the target machine. However, static translation is limited by the inability to discover all the code of a program since code and data are represented in the same way. For example, the targets of indirect transfers of control such as jumps on registers are sometimes hard to analyse statically. Therefore, a fall-back mechanism is commonly used: an interpreter. The interpreter processes any untranslated code at run-time and returns to translated code once a suitable path is found.

Dynamic binary translation can overcome the limitations of static translation, but at the expense of some performance. However, some optimisations that are not possible statically are possible dynamically (eg run-time constants propagation).

2.1 UQDBT framework

Figure 1 illustrates the UQDBT framework, which uses a basic block as the unit of translation (ie. as its granularity). The left-hand side recovers the semantic meaning of the machine instructions by a three-step process: decoding the binary file; decoding the machine instructions of the code segment; and mapping these instructions to their semantic meaning in the form of register transfer lists (RTLs). The processing of code is done at a different level of granularity (typically, one basic block at a time). The semantic of the source machine (Ms-RTL) is abstracted to a higher machine independent level (I-RTL) by the basic block translator. The right-hand side is the code generation process. The first time a basic block is translated, assembly code for the target machine is emitted and encoded to binary form. Simple optimisations that can be done quickly may be performed. Register caching at the basic block level is performed by UQDBT. The translated binary is run directly from the target machine's memory as well as being kept in a cache. A map of the source and target addresses of the program is stored. If a particular set of basic blocks is executed repeatedly, when the number of executions reaches a threshold, optimisations are performed dynamically to generate better code for that piece of code (a hot path). The system invokes the *optimiser* to determine the set of basic blocks that contributes to this hot path and to create a specialised version of the code in the hot cache. Different levels of optimisation are possible depending on the number of times the code is executed.

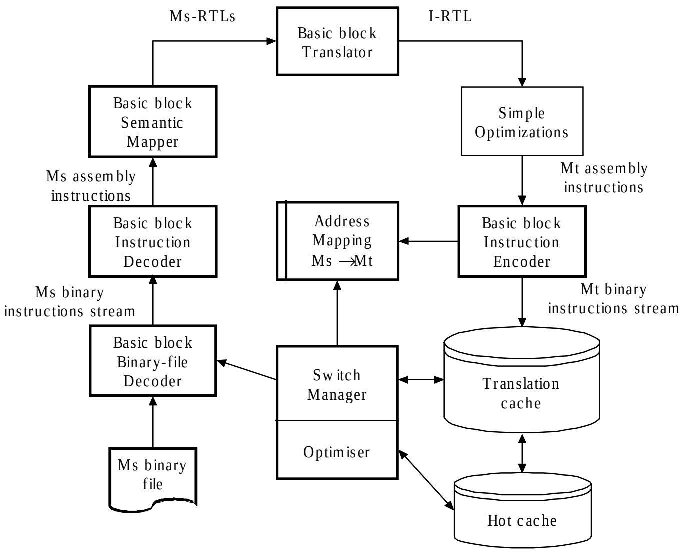

Figure 1. Dynamic binary translation framework

The processing of basic blocks is driven by the *switch manager*. The *switch manager* determines whether a new translation needs to be performed by determining whether there is an entry corresponding to a source machine address in the map. If an entry exists, the corresponding target machine address is retrieved and its translation is fetched from the cache. If a match is not found, the switch manager directs the decoding of another basic block at the required source address.

2.2 Machine-adaptability

UQDBT differs from other dynamic translators in that it can easily adapt to different source and target machines. Machine-adaptability is achieved by providing a clean separation of concerns - allowing machine-dependent information to be specified, and performing machine-independent analyses. The machine-dependent parts of the system are isolated and identified in the form of specifications. Through the use of specifications, a developer is able to concentrate on writing descriptions of properties of machines instead of having to (re)write the tool itself. These machine-dependent specifications can generate parts of the system automatically, which are integrated into UQDBT. It may also provide support for building components by providing a skeleton for the user to work on.

The translator itself is not bound to any hardware (unlike existing translators). UQDBT can support a variety of CISC and RISC machines at low cost through reusing the framework. New machines can be supported by writing their specifications and machine-specific modules. To provide for machine adaptability, extensions were made to Figure 1 which enable UQDBT to easily adapt to different source and target machines.

Machine-dependent information applies to three different levels of the system during decoding and encoding. These are:

1. binary-file format of the program,
2. syntax of the processor's machine instructions, and
3. semantics of the processor's machine instructions.

We have experimented with three different languages, reusing the SLED language and developing our own BFF and SSL languages:

- BFF: the binary-file format language supports the description of a binary-file's structure [15] and provides automatic generation of the loader code that decodes binary files. Currently, this is experimental work.

- SLED: the specification language for encoding and decoding supports the description of the syntax of machine instructions [18]. SLED is supported by the New Jersey machine-code toolkit [13], which provides partial support for automatically generating an instruction decoder and its components.

- SSL: the semantic specification language [16] allows for the description of the semantics of machine instructions. It provides the assembly transitions between RTLs and binary instructions.

3. HOT PATHS

Optimisations that are based on static analysis in traditional compiler technology can enhance the performance of the program, but often incur a heavy cost. In a dynamic environment, the choice of optimisations to be performed significantly affects the balance between time spent outside program execution and the effectiveness of the data collection process. It is important to ensure that the benefits of applying dynamic optimisation outweigh its cost. Sections of program code that are found to be executing very frequently are candidates for optimisation. Rather than trying to optimise everything that the translation comes across (as in a static system), recognising what to optimise is an important factor for improving speed in a dynamic system.

A hot path is a sequence of instructions that are frequently executed during a particular run of a program. Since most programs spend most of their time executing in small regions of code, being able to identify and optimise the hot path will often result in boosting execution speed.

3.1 Identifying a hot path

We use basic blocks as the base structure for our algorithm. A basic block is a sequence of instructions, which ends with a control transfer, ie. a branch, call or jump instruction. A branch into the middle of an already translated basic block creates a new translation starting at that point, which may duplicate parts of the translation. An edge is a directed pair that denotes the control flow from one basic block to another. For example, the edge *e* from basic block *a* to basic block *b* is denoted: $e = (a \to b)$

Each edge has a weight, which measures the number of times that a particular edge was traversed. We use these weights as a guide to find the hot paths of a program.

Let *BB* be the set of all basic blocks and *E* be the set of all possible flow edges in a program. If a program is stopped at a particular execution point, then the set of edges that are candidates for optimisation are ones with weights that exceed a predetermined value (minimum base value) that make it worthwhile for optimisation. The set of candidate edges is defined as:

$$Candidates = \{x \in E \mid Weight(x) \geq BaseCount\}$$

At run-time, a trigger is set for suspending a program's current execution so optimisation can take place. This is the *Trigger*, which is the minimum value for an edge to trigger optimisation. If an edge *t* reaches *Trigger*, then the edges of a hot path consist of edges reaching out from *t* that are also part of *Candidates*. Starting from *t*, we are only interested in the out edges of *t* that are hot. The set of out edges from *t* is defined as:

$$OutEdge(t) = \{x \in E \mid \forall a,b,c \in BB \bullet t = (a \to b) \wedge x = (b \to c)\}$$

From these out edges, we only pick the edges that are hot and build our next set of hot edges. We repeat this as follows:

$$HotSet(0) = \{t\}$$
$$HotSet(1) = \{x \in Candidates \mid \forall y \in HotSet(0) \bullet x \in OutEdge(y)\}$$
$$HotSet(2) = \{x \in Candidates \mid \forall y \in HotSet(1) \bullet x \in OutEdge(y)\}$$
$$\ldots$$
$$HotSet(n) = \{x \in Candidates \mid \forall y \in HotSet(n-1) \bullet x \in OutEdge(y)\}$$

The final set of hot edges derived from *t* that will make up the hot path is the union of all the hot sets:

$$HotEdgeSet(t) = \bigcup_{i=0}^{n} HotSet(i)$$

The upper value of n is limited, it is the smallest n such that:

$$HotSet(n) \subseteq \bigcup_{i=0}^{n-1} HotSet(i)$$

This ensures that for each new hot set found at step `y`, there exist some edges in this hot set `y` that are not part of previous `y-1` sets. The algorithm for finding *HotEdgeSet(t)* is expensive to compute if applied as is. In actual implementation, *HotEdgeSet(t)* is built incrementally by a recursive algorithm which adds edges that are not already traversed and have a minimum weight of *BaseCount*.

3.2 Creating a hot path

After the *HotEdgeSet(t)* is identified, optimisation takes place through a series of transformations on basic blocks that are part of the *HotEdgeSet(t).*

$HotBBSet(t) = \{x, y \in BB\}$, for all $(x \to y) \in HotEdgeSet(t)$

For each hot edge within the *HotEdgeSet(t)*, the control flow relationship between those basic blocks is duplicated and regenerated for specialisation. *HotBBSet(t)* is customised (optimised in ways that take advantages of program behaviours) by the relationships in *HotEdgeSet(t)* to create a new set of basic blocks in the hot path:

$HotPathBBSet(t) = \{customise(z)\}$, for all $z \in HotBBSet(t)$

with their new flow relationships

$$HotPathEdges(t) = \{e \in E \mid \forall x, y \in HotPathBBSet(t) \bullet e = (x \to y)\}$$

The code generated is a collection of customised basic blocks *HotPathBBSet(t)* linked by edges *HotPathEdges(t).*

For edges that are not part of the *HotPathEdges(t),* but are an out-edge of a *HotPathBB (element of HotPathBBSet(t)),* we create a *portal* out of the hot path back to the original unoptimised translation. An out-portal is a basic block that transfers control outside of the hot path. An out-portal from *HotPathBB* x is

$$OutPortal(x) = \exists y \in BB \bullet (x \to y) \notin HotPathEdges(t)$$

Then all the possible ways out of the hot path are the edges from a *HotPathBB* to their out-portals. The set of relationships leaving the hot path is:

$$OutPortalSet = \{e \in E \mid \forall x \in HotPathBBSet(t) \bullet e = (x \to Portal(x))\}$$

The edge coming into a hot path is the in-portal:

$InPortal = (x \to customise(x))$, and $(x \to y)$ is the edge that triggered the optimisation, and $y \in BB$.

Figure 2 shows the in-portal and out-portals of a hot path. The final hot path is defined as:

$$HotPath = HotPathEdges \cup OutPortalSet \cup \{InPortal\}$$

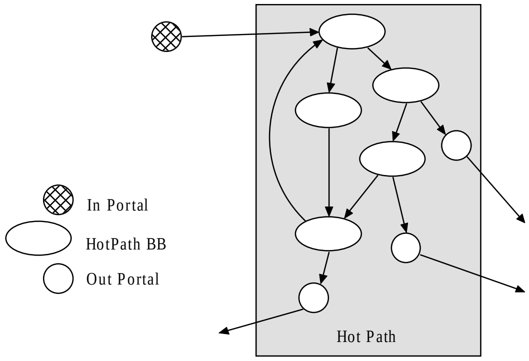

Figure 2. In-portal and out-portals of a hot path

3.3 Re-Optimisation of hot paths

Certain programs may exhibit behaviours of running in one section of code for some time then changing to another section. We call this a *staging behaviour*. A stage is a time frame in which a program expresses the same behaviour during that time. A program can have many stages. A change of stage suggests that a new hot path is likely to happen. For example, program execution leaves the current hot path and spends time in unoptimised blocks. If one of these edges becomes hot and hits the *Trigger,* the algorithm for finding hot paths is executed. A hot path is generated to reflect the new stage. The above algorithm works well for unoptimised basic blocks and can be extended to handle portals as well. We have four different scenarios depending on the properties of edge *e* that triggered the re-optimisation process.

A. If *HotEdgeSet(e)* does not contain any in-portals, then a completely new hot path needs to be created. Since there are no hot paths under consideration, the algorithm from the previous section remains unchanged.

B. If *e* contains an out-portal from hot path *h*, then it suggests that more basic blocks should be included with the current hot path containing *e*. The new extended hot

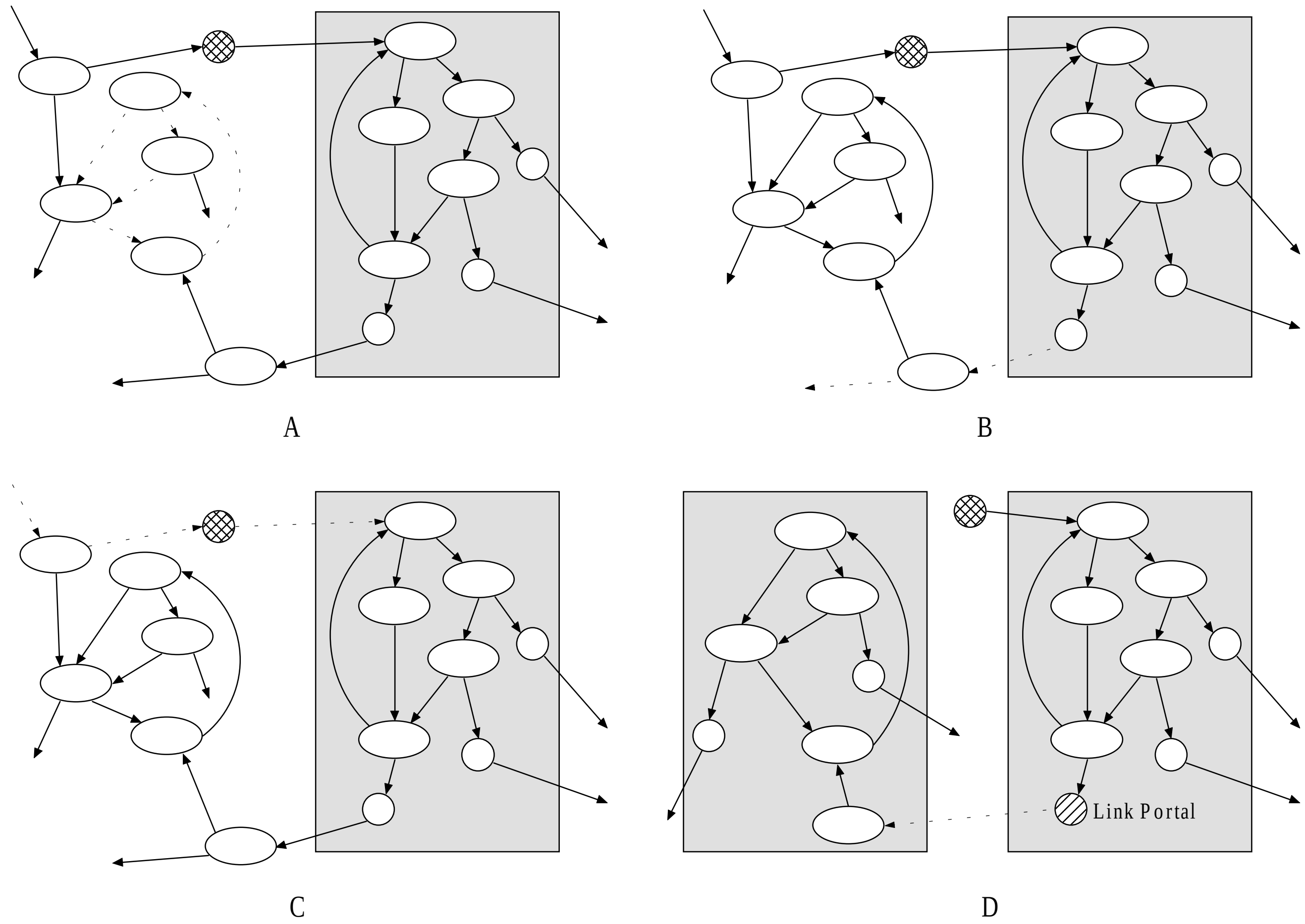

Figure 3. Re-optimisation of Hot Path scenarios

edge set (*HotEdgeSet'*) is the union of *HotPathEdges(t)* from hot path *h* and *HotEdgeSet*(*e*).

$$HotEdgeSet' = HotPathEdges(t) \cup HotEdgeSet(e)$$

The new *HotPathBBSet* is then built by customising basic blocks in the extended hot edge set – *HotEdgeSet'*.

C. If the hot edge set obtained from *e* contain in-portals to hot path *h*, then we need to consider how the transformation affects the relationship between hot path *h* and the unoptimised basic blocks. There is also the option of whether to traverse into hot paths via the in-portals. If traversing into hot paths is done, more edges are possible added by the algorithm as the out-portals may themselves be a start to their own hot edge sets. Each edge to an out-portal with a weight of *BaseCount* will invoke the algorithm to find their *HotEdgeSet*.

$$PortalSet = \{x \in OutPortals \bullet Weight(x) \geq BaseCount\}$$

For each edge *y* found in the *PortalSet*, *HotEdgeSet(y)* is calculated and added to the final extended hot edge set.

$$HotEdgeSet' = \bigcup_{y \in PortalSet} HotEdgeSet(y)$$

D. If *e* is a link-portal (an edge between two portals), then the two hot paths need to be merged. Basic blocks from the two hot paths can both come from the same original unoptimised block, but are duplicated in the final merged hot path. For example, if edge A → E is optimised in hot path *h1* and edge A → F is optimised in hot path *h2*, block A is duplicated in the final merged hot path so that the relationship from A → E and A → F remains customised. The algorithm already handles duplicated blocks in that A from *h1* is different to A from *h2*. The transformation customise(A) in *h1* is different to the transformation customise(A) in *h2*.

Figure 3 show the 4 different scenarios when re-optimisation takes place. Dotted arrows indicate edges that are hot in the new stage.

4. OPTIMISATIONS IN UQDBT

Optimisations can be applied as early as generating code at translation time (the simple optimisation phase of UQDBT), but the main optimisation is done during program execution. Dynamic optimisations involve examining its execution behaviour and choosing the most relevant code areas for optimisations to achieve the best performance. Apart from register caching that is performed during the initial code generation phase, all other optimisations by UQDBT are dynamic. As described in the previous section, identifying hot edge sets and building hot paths give information on what to optimise.

4.1 Instrumentation

The *optimiser* component of UQDBT is triggered at run-time to identify a hot path. It profiles a program's control flow during execution. There are two possible profiling techniques that could be used: node weight and edge weight profiling. Node weight profiling measures the frequency that a node (basic block) is executed, while edge weight profiling measures the frequency of flow control taken between two nodes. We use edge weight profiling since it better reflects the connection between the basic blocks during execution. An edge weight is a pair of basic blocks and a frequency counter called the weight, initially set to zero at code generation time. Each time execution takes place through an edge, it increments the weight of that edge. Figure 4 shows the instrumentation code added to the generated translation for a two-way basic block. Stubs are created for each out-edge to other basic blocks which may or may not have been translated. The stub is also a place for adding instrumentation. Initially the last jump at the end of a stub directs execution to the *switch manager*. It is later patched by the run-time system when the program takes this path, which triggers the translation of the appropriate code.

The *optimiser* is invoked by the stub code when its weight reaches a predefined threshold – the *Trigger*. Program execution is suspended at this point, and the *optimiser* determines the set of flow edges that contributes to the execution hot path – the *HotEdgeSet*. The hot path is formed through re-generating and customising each basic block found in the *HotEdgeSet*. The code for the hot path is emitted to the hot cache. The algorithm for identifying a hot path is implemented as a recursive function that takes an edge as a parameter and updates the *HotEdgeSet* incrementally. The out-edges of the current edge under consideration are examined and follows paths that are potentially hot (within the path inclusion threshold – *BaseCount*).

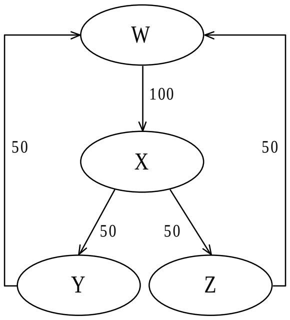

Figure 5: Single level evenly executed two-way branch

The *BaseCount* should be set to a value just below 50 percent of *Trigger* to compensate for any single level evenly executed two-way branches. Figure 5 show an evenly executed two-way branch from X to Y and X to Z. When edge WX reaches *Trigger* (weight of 100), we also want to include edges XY, XZ, YW and ZW in the hot path. For two or more levels of evenly executed branches, more paths can be considered through modification to the algorithm to allow comparing of weights between out-edges. For these types of programs, there is no single hot path that is worth optimising.

But most programs tend to take a particular path most of the time instead of evenly executing between paths. Figure 6 is a snap shot of a typical program at the point where the *optimiser* is invoked. The *Trigger* is set to 50 and the *BaseCount* is 20, which is 40 percent of the *Trigger*. The edge A to B is the edge that triggers the optimisation to take place. The other two edges, B to E and E to A is also identified as part of the *HotEdgeSet*.

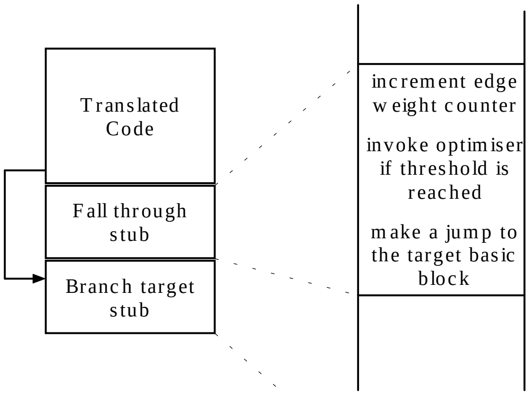
Translated Code
Fall through stub
Branch target stub
increment edge weight counter
invoke optimiser if threshold is reached
make a jump to the target basic block

Figure 4. Instrumentation for a two-way node

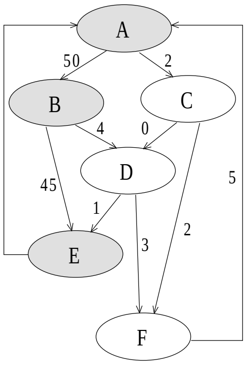

Figure 6. Weight graph before optimisation

4.2 Improving code locality

The discovery of hot paths not only allows the *optimiser* to identify which section of the program is most beneficial for optimisation; it can also improve the locality of the translated code through moving and merging of basic blocks. By moving the frequently executed basic blocks together, caching is improved. Figure 7(a) shows the code initially generated by the translator. Each generated basic block A, B and E occupies a space of its own and lies some distance from the others. By moving those basic blocks together, the locality of the basic block translations are improved and hence better utilisation of caching is achieved. An added bonus to having localised basic blocks together is that there is the opportunity for reduced control flow between them. This can be achieved by merging the basic blocks together, thus eliminating unnecessary control flow transitions. Figure 7(b) shows the result after moving and merging the basic blocks together.

After the *optimiser* determines the hot edge set, each basic block is retranslated and put into the hot path. In the example above, the basic blocks A, B and E are retranslated. Retranslation is necessary as the *optimiser* uses the information gathered about the program's behaviour to create new translations that better reflect the program's control flow, thus improving performance. The hot path itself is a special version of a basic block in that it allows control flow exits (out portals) from anywhere within the block instead of only at the end. The starting basic block is patched and redirects program execution to the beginning of the hot cache. All other blocks remain unchanged, so code that branches to these blocks is unaffected. The hot path can be seen as a specialised version of the execution path consisting of the customised, merged basic blocks. Figure 8 shows the state of the translated code after generation of a hot path. The original block A is patched with a jump to the beginning of the hot path. A*, B* and E* are retranslations of the original basic blocks. Pc and Pd are the out portals connecting back to the original basic blocks C and D.

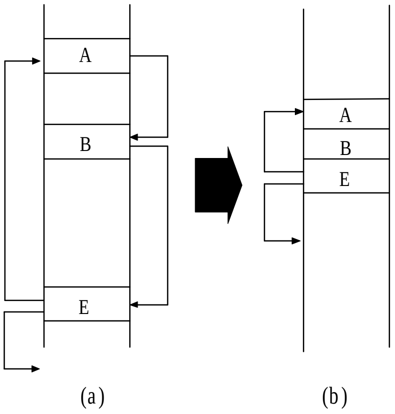

Figure 7. Moving and merging of frequently executed basic blocks

4.3 Path prediction feedback

The hot path is constructed by the *optimiser* through retranslation of each basic block found in the *HotEdgeSet*. The customising process may modify the branch instruction to improve the locality of the code through path prediction from data gathered during the program run. For example, assume the following piece of SPARC branch code:

```
             cmp %o0, 100
             ble .LabelA
             nop
fallthrough:
             ...
             ...
.LabelA:
             ...
```

If during program execution the branch is taken most of the time, it is worthwhile inverting the branch instruction to make .LabelA the fall through, so that the more frequently executed code is closer together. Feedback of information is provided by UQDBT, which allows the *optimiser* to make the necessary modifications.

4.4 Other optimisation opportunities

The moving and merging of smaller basic blocks into the hot path can provide further optimisation opportunities, similar to the inlining. In UQDBT, this proves to be a critical element for improving performance in the hot path. During the initial translation process (before the *optimiser* is called), care must be taken to ensure that the control flow of the translation is the same as the original program. Extra housekeeping is necessary to ensure that the value of the virtual registers (locations that represent registers of the source machine) reflects the same state as the original program at each exit of a basic block. This is critical because the next target address may not yet be translated and could potentially call other components of the system like the *switch manager*. But within the hot path the

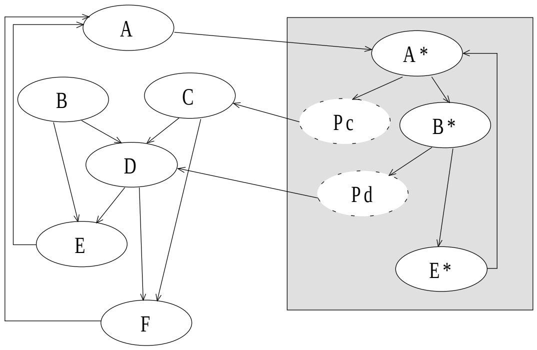

Figure 8. Program state with hot path

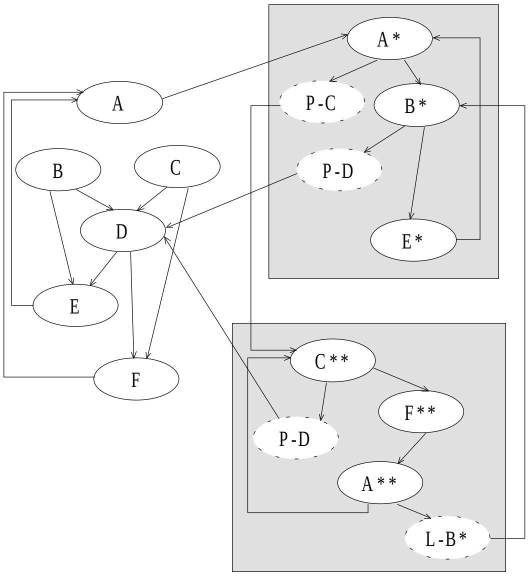

Figure 9. Re-optimisation from an out portal

housekeeping that was needed previously is no longer valid since the hot path is like a super basic block. The basic blocks that spread across the hot path are now part of the global structure. This leads to better utilisation of the control flow during translation and opens up the next stage of optimisation, which can be perceived as inter-basic block rather than intra-basic block optimisations. With a larger set of data that is gathered within these basic blocks, the register caching strategy is vastly improved as more code is revealed through path feedbacks.

While all these optimisations are happening within the hot path, branch exits out of the hot path need to link to translations that were initially generated or are yet to be generated. Any infrequently executed code and any housekeeping code needs to be moved outside of the hot path. UQDBT implements this by creating a portal that links between the hot paths and any other blocks coming out of it. The portal may contain any housekeeping code and instrumentation code for triggering re-optimisation, for the case when an exit from the hot path becomes hot.

4.5 Re-optimisation

The execution behaviour of a program can change from one state to another. A program can take a particular path for a period of time, and then choose a different path later on. During the execution life of a program, there will be times for which the program shows the same behaviour and having a particular state (executing the same path). Therefore, finding only one hot path may not be optimal in many cases. To fully utilise the translation system, it needs to identify these states and add the new hot paths to the existing hot cache. Figure 9 show the new hot path generated when the out portal to C from Figure 8 is hot. The out portal is a bridge from A* to C, and the following edges are also found to be hot: C to F and F to A. This shows that the program is possibly changing to another stage of execution. The new hot path CFAC is generated and customised. The link portal L-B* converts register states between the hot paths and takes care of any house keeping code. Instrumentation code also exists as part of the link portal.

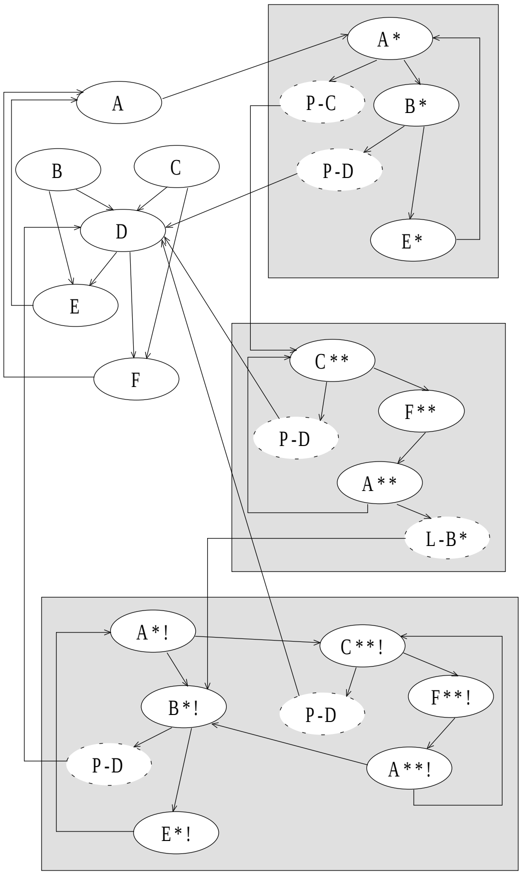

Figure 10. Re-optimisation from a link portal

A program may decide to cycle between hot paths. Figure 10 shows the output of merging two hot paths to reduce the control transfer between them. The first hot path customises the path ABEA while the second customises CFAC. Notice that the basic block A is re-generated twice to keep the original specialised hot paths. The contents of A*! and A**! are different since *customise(A)* in the first hot path is different to *customise(A)* in the second.

4.6 Performance and overhead

The current implementation of UQDBT can translate programs between Intel x86 and SUN SPARC Solaris-based binaries. Instrumentation code costs up to 13 SPARC (fewer for x86) instructions for each control flow stub in a basic block, though only half of these are usually executed at run-time. This incurs a 5 to 20 percent execution overhead with profiling turned on.

The time spent in the *optimiser* is proportional to the length of the hot path. The size of a hot path ranges from just 2 basic blocks to the maximum free space of the hot cache (changeable by the user). When the program changes between stages, new hot paths tend to get larger due to code replication. Code in the hot path is not profiled, hence no overhead is incurred once it is optimised.

Programs that are optimised using edge weight profiles run up to 15 percent faster than non-profiled translations (non-profiled translations currently run 2.5 to 7 times slower than native programs compiled on the target machine). Programs that do not optimise well simply incur the cost of profiling in translations, but not in hot path code. This ensures that frequently executed code does not get penalised. These results are based on small/medium sized programs that are less than 100KB in size. Micro-benchmark results were obtained using a Pentium MMX 250 MHz machine and an UltraSparc II 250 MHz machine, both running the Solaris operating system.

The test programs showed in the tables are:

- Sieve 3000 (prints the first 3,000 prime numbers),
- Fibonacci of 40, and
- Mbanner (prints the banner for the "ELF" string 500,000 times).

Sieve mainly contains register to register manipulation, while Fibonacci has a lot of recursive calls and Mbanner has a lot of stack operations (on Pentium) and accesses to an array of data which requires byte-swapping when translating between different endian machines.

Tables 1 show the times of translation, optimisation and execution of programs under UQDBT (from Pentium to SPARC), compared to natively gcc -O0 compiled programs. The source programs were also -O0 compiled on the Pentium machine. Column 2 shows the startup time that is needed before the actual translation takes place. Column 3 shows the total time spent decoding the source instructions, transforming them to I-RTLs and generating the final SPARC code. Columns 4 and 5 show the total execution time without hot path optimization, while columns 6, 7 and 8 is with hot path optimisation. Both groups show the times with and without register caching (the allocation of target registers to virtual registers and sub-expressions). Register caching yields between 10 to 50 percent performance gain. Column 6 shows the time spent in the *optimiser* which generates hot paths to the hot cache. Column 9 is the natively compiled gcc version of the same program on SPARC. From the table, hot path optimisations can render an additional 15 percent improvement over non-profiled translations. The figures also suggests a 2.5 to 6 times slowdown when running programs using UQDBT with hot path optimisation. Translations from SPARC to Pentium will exhibit the same benefits from these optimisations and results can be expected to be very similar.

4.7 Future work

One of the major difficulties with translation from one machine to another is the effect of processor condition codes. In CISC machines like x86, many instructions set the condition codes as a side effect. Some of these have no effects and may be overwritten by the later instructions. When translating to another machine, it is desirable to eliminate any unnecessary simulation of the condition code from the original machine, especially when generating code for the hot cache. Determining whether the effects of a condition code can be removed involves examining the code to see if there are any uses that follow. This process is difficult to complete in dynamic systems as it can involve analysing across basic block boundaries, some of which may not even be generated at the time of the analysis. Although full removal of condition code is impractical dynamically, it is possible to provide partial removal. It is a question of how far the analysis process

			Without Hot Paths		With Hot Paths			
Test programs	Startup time	Translation time	Execution time without reg caching	Execution time with reg caching	Optimisation time	Execution time without reg caching	Execution time with reg caching	Native gcc compiled
Sieve3000	0.54	0.14	98.25	73.14	0.16	80.98	66.29	29.22
Fibonacci	0.54	0.10	186.17	154.97	0.11	147.56	133.69	41.18
mbanner	0.52	0.34	219.01	146.28	0.37	146.22	126.28	22.85

Table 1: Pentium to SPARC translation (second)

can go before it decides to be conservative and how to restore the original states of the condition codes when exiting from the optimised code cache. To ensure program correctness, the current implementation of UQDBT conservatively emulates all side effects of condition codes on the target machine. This results in 50% more code in the final translation for every source instruction that affects the condition codes.

5. RELATED WORK

The early binary translators like Digital's VEST and mx [1], Apple's MAE [2] and Digital's Freeport Express [3] were all static. In recent years, hybrid translators like Executor by Ardi [5] and Sun’s Wabi [6] mix translation with emulation. Embra [7] and Shade [17] use dynamic translation techniques. Le[8] also investigated in out-of-order execution techniques in dynamic binary translators. Other forms of systems that use dynamic translation to process non-native code are Just-In-Time (JIT) compilers for Java [11,12].

Dynamic compilation is generally found in dynamic compilers such as DyC[9] and tcc[10]. Systems such as these either require the programmer to annotate special parts of the program for dynamic compilation at run-time, or the program must be written in a specialised language. In dynamic translation, the optimisation process is transparent to the user.

Profiling the program’s behaviour during execution can assist in identifying hot paths for optimisation. Digital’s FX!32 [4] performs native optimisations from profile data that was collected during an initial run of the program through emulation. The program is then translated and optimised offline. The optimisation process is static and its benefits are only available during later re-runs of the program. In UQDBT, the optimisation is done at run-time, hence it is effective from the first run.

Dynamic optimisers like Dynamo [21] and Wiggins/Redstone [22] take a native binary program and optimise it on the same machine. In contrast, dynamic translators take code from one machine and translates it to a different machine. Dynamic optimisers often rely on the underlaying hardware to provide efficient implementation of instruction speculation to identify hot paths. Because of this, their overhead is small compared to UQDBT, where instrumentation code is generated on the fly using edge weight profiling which is hardware independent.

The way Dynamo identify hot sections of code is by speculation through instruction interpretation, while Wiggins/Redstone is based on sampling. Both systems apply optimisations to a hot trace, built from the time when the start of a trace is initiated. The idea is that if a branch instruction is identified to be the start of a trace, the set of instructions executed immediately after is likely to be hot as well. Sometimes this can cause the program to pick the wrong path as the trace. For example, in Figure 6, edge AB is the start of a trace when it hits a count of 50. But if at that time the program takes the path BE instead, then the trace that is built will be different to the hot path. Profiles can identify a hot path correctly because it contains complete history information about program execution paths. Another situation where profiling excels is when encountering evenly executed branches. For example, in Figure 5, a hot trace will just pick one side of the path.

6. CONCLUSION

Hot paths are collections of code that are identified to be frequently executed over a period of time. Optimising hot paths can improve speed of an executing program. UQDBT is capable of recognising the staging behaviour of a program and identifying hot paths.

Optimisations performed in UQDBT are generic and can be applied to various different types of machines. The system does not rely on the underlying machine to provide support for profiling. Instead, UQDBT employs edge weight profiling by instrumenting translations. The main optimisations are achieved by better utilisation of caches and improving localisation of code within hot paths. Re-optimisation is performed when a program changes execution stages. Although using edge weight profiling incurs some run-time overhead and is slower than sampling, the information collected is complete and very effective for finding hot paths. As a result, UQDBT not only supports multiple machines but optimisations can also be performed for multiple machines.

7. ACKNOWLEDGMENTS

The authors wish to thank Mike Van Emmerik and Dan Johnston for their helpful discussions in implementation and testing strategies. This work is part of the University of Queensland Binary Translation (UQBT) project. More information can be obtained about the project by visiting the following URL:

http://www.csee.uq.edu.au/csm/uqbt.html.

8. REFERENCES

1. R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson. *Binary translation*. Communications of the ACM, 36(2):69-81, February 1993.
2. Apple Corporation. *Macintosh application environment*. http://www.mae.apple.com/, 1994.
3. Digital. *Freeport express.* http://www.digital.com/amt/freeport/, 1995.

4. R.J. Hookway and M.A. Herdeg. *Digital FX!32: Combining emulation and binary translation*. Digital Technical Journal, 9(1):3-12, 1997.

5. ARDI. *Executor Internals: How to Efficiently Run Mac Programs on PCs.* http://www.ardi.com/MacHack/machack.html, 1996.

6. SunSoft. *Wabi*. http://www.sun.com/sunsoft/Products/PC-Integration-products/, 1994.

7. Emmett Witchel and Mendel Rosenblum, *Embra: Fast and Flexible Machine Simulation*. The proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems, Philadelphia, 1996.

8. Bich C. Le. *An out-of-order execution technique for runtime binary translators*. In Proceedings of the 8th international conference on Architectural support for programming languages and operating systems, pages 151-158, San Jose, CA. Oct 1998.

9. B. Grant, M. Mock, M. Philipose, C. Chambers, S.J. Eggers. *Annotation-Directed Run-time Specialization in C*, Symposium of Partial Evaluation and Semantic-Based Program Manipulation, June 1997.

10. Massimiliano Poletto, Dawson R. Engler and M. Frans Kaashoek. *tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation*. In PLDI '97. Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation, pages 109-121, Las Vegas, NV, June 1997.

11. Sun, *Java JIT compiler*, http://www.sun.com/solaris/jit.

12. Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh and James M. Stichnoth. *Fast, effective code generation in a just-in-time Java compiler*, In PLDI '98, Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation, pages 280-290, Montreal, Canada, June 1998.

13. Norman Ramsey and Mary Fernández. *The New Jersey Machine-Code Toolkit*. Proceedings of the 1995 USENIX Technical Conference, pages 289-302, New Orleans, LA, January 1995.

14. C. Cifuentes, M. Van Emmerik and N. Ramsey, *The Design of a Resourceable and Retargetable Binary Translator*. In Proceedings of the Working Conference on Reverse Engineering, pages 280-291, Atlanta, USA, Oct 1999. IEEE CS Press.

15. D. Ung and C. Cifuentes. SRL – *a simple retargetable loader*. In Proceedings of the Australia Software Engineering Conference, pages 60-69, Sydney, Australia, Sept 1997. IEEE CS Press.

16. C. Cifuentes and S. Sendall. *Specifying the semantics of machine instructions*. In Proceedings of the International Workshop on Program Comprehension, pages 126-133, Ischia, Italy, 24-26 June 1998, IEEE CS Press.

17. B. Cmelik and D. Keppel. Shade*: A Fast Instruction-Set Simulated for Execution Profiling*. SIGMETRICS, Nashville, TN, 1994.

18. Norman Ramsey and Mary Fernández. *Specifying representation of machine instructions*. ACM Transactions of Programming Languages and Systems, 19(3):492-524, 1997.

19. C. Cifuentes, M. Van Emmerik, D. Ung, D. Simon and T. Waddington, *Preliminary Experiences with the Use of the UQBT Binary Translation Framework*. In Proceedings of the Workshop on Binary Translation, Newport Beach, USA, Oct 1999. Published in Technical Committee on Computer Architecture News, pages 12-22, Dec 1999, IEEE CS Press.

20. D. Ung and C. Cifuentes, *Machine-Adaptable Dynamic Binary Translation.* In Proceedings of the Workshop on Dynamic and Adaptive Compilation and Optimization, pages 37-47, Boston, Massachusetts, USA, Jan 2000, ACM SIGPLAN.

21. V. Bala, E. Duesterwald and S. Banerjia, Dynamo: A *Transparent Dynamic Optimization System.* Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, Vancouver, Canada, 2000.

22. D. Deaver, R. Gorton, N.Rubin, *Wiggins/Redstone-An On-line Program Specializer*, Hot chips 11 – A symposium on High performance chips, Stanford University, CA, USA, Aug 1999.