Paralelização Dinâmica de Executáveis utilizando Trace Trees

João Paulo Fernandes Ventura
Universidade Estadual de Campinas
Av. Albert Einstein, 1251 CEP 13083-970
Campinas/SP, Brasil
ra024115@students.ic.unicamp.br

ABSTRACT

A barreira térmica em microprocessadores parece sinalizar o fim dos grandes aumentos periódicos da freqüência dos processadores. Impedida de continuar o ganho de desempenho no mesmo ritmo da miniaturização dos transistores, a indústria aposta em arquiteturas com vários núcleos de processamento (*cores*) integrados em uma única pastilha de silício. Esse artigo apresenta a combinação de três técnicas para exploração de paralelismo em códigos binários executáveis legados cujos algoritmos são seqüenciais. [1]

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Metrics—*Performance Measures*

General Terms

Measurement Performance Verification

Keywords

Architecture, Compilers, *Just-in-Time*, Multicore

1. INTRODUÇÃO

A barreira térmica em microprocessadores parece sinalizar o fim dos grandes aumentos periódicos da freqüência dos processadores. Impedida de continuar o ganho de desempenho no mesmo ritmo da miniaturização dos transistores, a indústria aposta em arquiteturas com vários núcleos de processamento (*cores*) integrados em uma única pastilha de silício. Estas arquiteturas permitem continuar a aumentar a densidade de uso do silício de tal modo que, dentro de algumas décadas, será possível confeccionar processadores com mais de 1000 *cores*[1].

[1]Este artigo é um trabalho realizado durante a disciplina de pós-graduação MO401 Arquitetura de Computadores I. Ela foi lecionada pelo professor Paulo César Centoducatte no Instituto de Computação da Universidade Estadual de Campinas durante o primeiro semestre de 2008.

As duas maiores dificuldades dessa arquitetura dizem respeito a como programá-las eficientemente e como obter ganhos de desempenhos em programas legados. A utilização de *threads* e memórias transacionais são exemplos de propostas para solucionar o primeiro problema, enquanto o segundo permanece em aberto. Programas cujos algoritmos são seqüenciais costumam gastar 80% de seu tempo em 20% de seu código, usualmente correspondendo a laços interiores. Esses trechos mais executados (*traces*) podem ser detectados sem grande esforço computacional através de *Trace Trees*[5][3], otimizados e paralelizados dinamicamente[9].

Esse artigo primeiramente apresenta uma rápida evolução dos processadores, as limitações do *instruction level parallelism* (ILP)[7][8] a técnica de *Trace Trees* (TT) e como ela permitiria a detecção dos caminhos críticos de execução do programa. Em seguida são apresentados os resultados dessa técnica aplicada em compiladores *Just-in-Time* (JIT) em máquinas virtuais executadas em processadores *multicore*. Finalmente é apresentado como um compilador dinâmico paralizante poderia utilizar TT.

2. MOTIVAÇÃO

Nos últimos 40 anos, a Lei de Moore[1] se manteve absoluta. Segundo ela, a quantidade de transístores inseridas, a um custo mínimo, em uma mesma pastilha de silício dobraria a cada vinte meses. Isso se refletiu em uma evolução acelerada do poder dos microprocessadores, permitindo adquirir hoje computadores pessoais com o mesmo desempenho de supercomputadores de 10 anos atrás. Mas ao contrário do que Gordon Moore acreditava, a dissipação de calor tornou-se a maior barreira desse aumento de desempenho, principalmente o crescimento da freqüência dos processadores. Atualmente a indústria aposta em arquiteturas com múltiplos núcleos de processamento (*cores*) mais simples, integrados em um único *wafer*, operando em freqüências menores e dissipando menos calor. Logo se supormos que a quantidade de transistores em um *cores* permanece constante, de acordo com a Lei de Moore a quantidade de núcleos em um processador dobraria a cada ano. Desse modo dentro de algumas décadas existiriam microprocessadores com mais de 1000 *cores*[1]!

Essas arquiteturas possuem dificuldades inerentes no desenvolvimento de *sofwares* que poderiam levar a sua subutilização. Dependências de dados e de controle diminuem a quantidade de *instruction level paralellism* que compiladores são capazes de obter de programas seqüencias, logo para

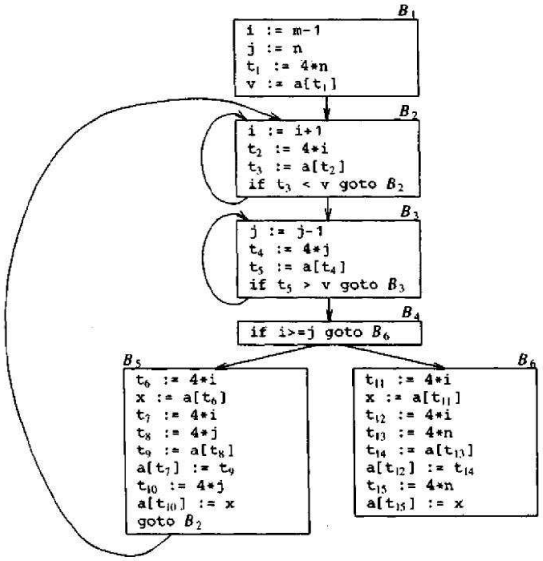

Figure 1: Exemplo de CFG.

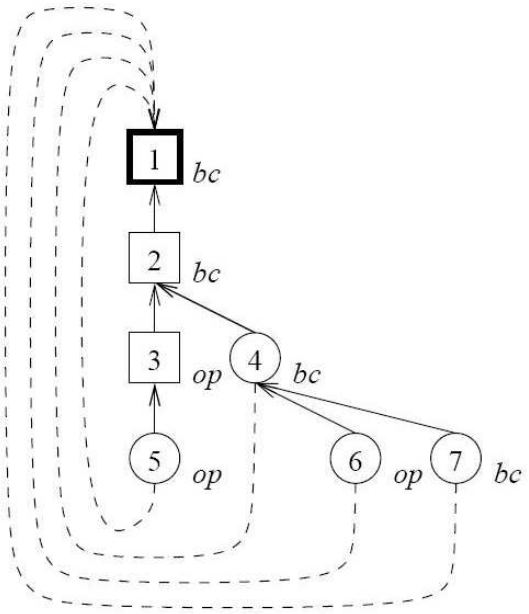

Figure 2: Exemplo de *trace tree*.

uma eficiente utilização desses novos modelos é necessário uma maior granularidade de paralelismo como processos e *threads*. Para que compiladores possam explorar eficientemente *thread level paralellism* (TLP)[6], é o programador precisa explicitar os segmentos de código paralelo. Técnicas como programação em *threads* associada à memórias transacionais são promissoras quanto ao desenvolvimento de novos programas eficientes, mas paralelizar *softwares* legados permanece um problema em aberto.

Programas cujos algoritmos são executados sequencialmente gastam 80% do tempo em apenas 20% de seu código, sendo que usualmente esse percentual corresponde a laços interiores. Essas são as principais fontes das quais compiladores poderiam extrair TLP e poderiam ser facilmente detectadas através de *Trace Trees*[5].

2.1 Trace Tree

Sun Java Virtual Machines (JVM) originalmente apenas interpretavam *bytecodes* compilados a partir do código fonte do programa. Compiladores *Just-in-Time* (JIT) surgiram para tornar esses programas mais rápidos, compilando as regiões críticas mais executadas (*HotSpots*). Esses compiladores tradicionalmente determinam e analisam essas regiões através da construção de um grafo direcional $G_{CFG} = (B, E)$ denominado *Control-Flow Graph* ou CFG[2] mostrado na figura 1. Os nós $b_i \in B$ (*Basic Blocks* ou BB) são instruções executadas seqüencialmente. Não é permitido que saltem para instruções do BB exceto a primeira, e não existem instruções que saltem para fora do bloco exceto a última. As arestas $e_i \in E$ representam desvios (in)condicionais. As informações provenientes dessas análises permitem que algoritmos de otimização sejam aplicados antes da emissão de código nativo.

Ao invés de construir uma representação para todo o programa, poderia ser utilizada uma abordagem "preguiçosa" que representasse apenas as partes mais executadas na forma de um grafos direcionais $G_{TT} = (N, P)$ denominados *Trace Trees*, mostrada na figura 2. Os nós $n_i \in B$ são instruções e o conjunto de arestas $(b_i, b_j) \in P$m, de modo que a instrução n_j é executada imediatamente após a instrução n_i. Definindo $\Gamma^1_{TT}(n_i)$ e $\Gamma^{-1}_{TT}(n_i)$ respectivamente como os sucessores e predecessores imediados de n_i. Instruções classificadas como *op* possuem no máximo um sucessor imediato ($|\Gamma^1_{TT}(n_i)| \leq 1$), sendo adições e multiplicações alguns exemplos. Uma instrução não é chamada de *anchor* se ela não possue nenhum predecessor ($\Gamma^{-1}_{TT}(n_i) = \varnothing$). Alguns nós n_i podem podem possuir arestas que apontam para um *anchor*, mas ainda assim teria $\Gamma^{-1}_{TT}(n_i = \varnothing)$ pois arestas de retorno (*back-edges*) são consideradas implícitas por não serem expressas no conjunto P. Esses nó são classificados como folhas da TT, podendo ser instruções *op* (*non-branching*) ou *bc* (*branching*).

As ávores são construídas incrementalmente com o decorrer da interpretação do programa de modo apenas instruções freqüentemente executadas são inseridas na TT. Inicialmente o compilador analisa a freqüência de saltos para uma instrução n_i, marcando a como *loop header* e como um *potential anchor*. Após exceder um número de execuções, n_i passa a ser um *actual anchor* e é inserido na *trace tree*. Nem todas as arestas dectectadas são inseridas na *trace tree*. Instruções interpretaras que geram excessões e acessos excessivos à memória abortam a criação de uma árvore. Caso esteja demasiadamente grande, seu cálculo também é interrompido.

O lado esquerdo da figura 3 mostra um *trace* primário (2,3,5). Um *trace* secundário (2,4,5) é detectado e inserido na TT inicial, compartilhando a instrução 2. A dominância da quantidade de execuções é o fator determinante para decidir qual *trace* será inserido primeiro, consequentemente tornando-se o primário. Os nós 2, 4 e 5 são classficicados como *side-exits* da *trace tree*. *Side-exits* são instruções de saltos que podem

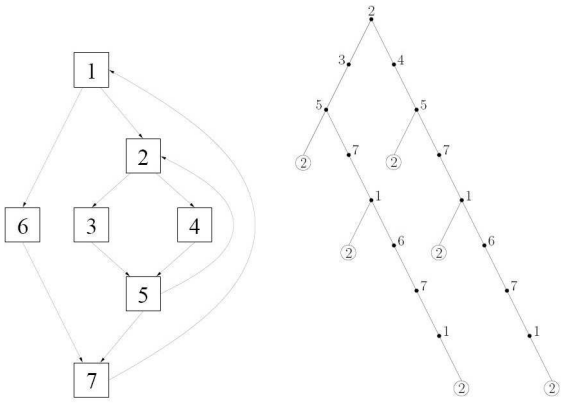

Figure 3: Laço interno em uma *trace tree*.

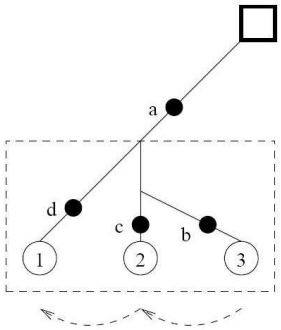

Figure 4: Emissão de código em uma *trace tree*.

gerar caminhos não cobertos pela *trace tree*. Se a instrução salta para outra que faça ou venha fazer parte da TT, então o *trace* correspondente é inserido na representação, como (2,4,5). Caso contrário, como o nó 5, o *trace* alcançado a partir dela não é inserido. Esse ponto pode corresponder a um salto do modo compilado para o modo interpretado na execução do programa ou o marcador de início de uma nova *trace tree*.

Como os *actual anchors* são determinados estatisticamente e os laços internos são os mais executados, os *headers* desses são inseridos na *trace tree* como *anchors*. Os percursos correspondentes aos laços externos são inseridos como *traces* secundários do laço interno, desse modo a TT parece virada do avesso em relação ao CFG correspondente como mostra a figura 3. Nessa figura as *back-edges* foram substituídas por arestas que apontam para cópias do *anchor*.

Observando a figura 3, seria desejável que o compilador emitisse a instrução 2 de modo que ela fosse compartilhada por 3 e 4. Portanto a emissão de código nativo deve ser feita a partir das folhas da árvore. Contudo é necessário que todos os usos de variáveis seja observado antes de suas definições de modo a garantir que uma definição sem uso seja desnecessariamente emitidada. Portanto a árvore deve ser percorrida em pós-ordem, garantindo também que as instruções dos *traces* secundários sejam emitidas antes do primário, como mostra a figura 4.

2.2 Implementação da Trace Tree

O primeiro estudo apresentado por [5] apresenta uma implementação da *trace tree* em uma máquina virtual *open-source* chamada JamVM. O segundo estudo [3] mostra uma implementação em Java da mesma técnica na forma de um *pipeline*. Os dados que fluem através do *pipeline* são classes passadas como mensagens (Figura 5), podendo ser instruções ou mensagens de controle que indicam o ínicio e o fim de uma árvore. As instruções que compõem a *trace tree* são basicamente classificadas como unárias ou binárias.

O *pipeline* é iniciado pela classe *Clone*, responsável por enviar cópias das intruções ao invés das originais. Isso garante que essas não sejam corrompidas no processo de otimização. Como a árvore é percorrida em pós-ordem, as instruções precisam ser invertidas pela classe *Reverse* para corresponder a correta ordem de execução do programa. Entre esse ponto e a emissão do código nativo pela classe *Assembler*, várias classes realizam repetidamente diversas análises e, a partir dessas, otimizações de código (exemplo: *invariante code analysis, constant propagation, common subexpression elimination,* etc...). O interessante dessa implementação é que todas essas classes implementam a interface *Runnable* em Java, permitindo que a compilação como um todo seja realizada paralelamente a interpretação do programa ou ainda realizar cada estágio do *pipeline* como uma *thread* independente. Por isso são necessárias duas barreiras para garantir a sincronização entre as *threads*.

Os resultados das modificações na JamVM e do compilador paralelo em Java são apresentados na seção 3.

2.3 Compilador Dinâmico Paralelizante

O terceiro estudo analisado apresenta um modelo de compilador que recebe como entrada um código binário executável (Figura 12). O compilador realiza uma análise estática do código que compõem o programa. Ao invés de analisar todos os blocos básicos que compõem o CFG do programa, ele utiliza uma menor granularidade quebrando o CFG em subgrafos chamados de *superblocos*. Basicamente esses superblocos são CFG menores que possuem um único ponto de entrada que domina todas as saídas do grafo. O compilador agrega a essa estrutura informações como uso e definições de variáveis, e a árvore de dominâcia dos blocos básicos que compõem o superbloco.

Uma vez detectados estaticamente os *HotSpots* desses superblocos, o compilador converte o programa para uma representação intermediária. Essa representação é recompilada dinamicamente por uma pequena camada de *software*, que paraleliza ou vetorializa os laços mais executados. Os laços não podem possuir dependências entre as iterações e precisam possuir limitantes fixos para as variáveis de indução para poderem ser otimizados.

Essa técnica pode ser aplicada em microcomputadores processadores com *simultaneous multithreading*, processadores *multicore* ou com vários microprocessadores. Usualmente os laços são divididos de acordo com a quantidade de unidades de processamento. Devido esse fato, laços com limitadores constantes dos iteradores são preferenciais. Uma alternativa para laços de número variáveis de iterações seria quebrá-las em vários blocos de tamanho constante. Porém é necessário

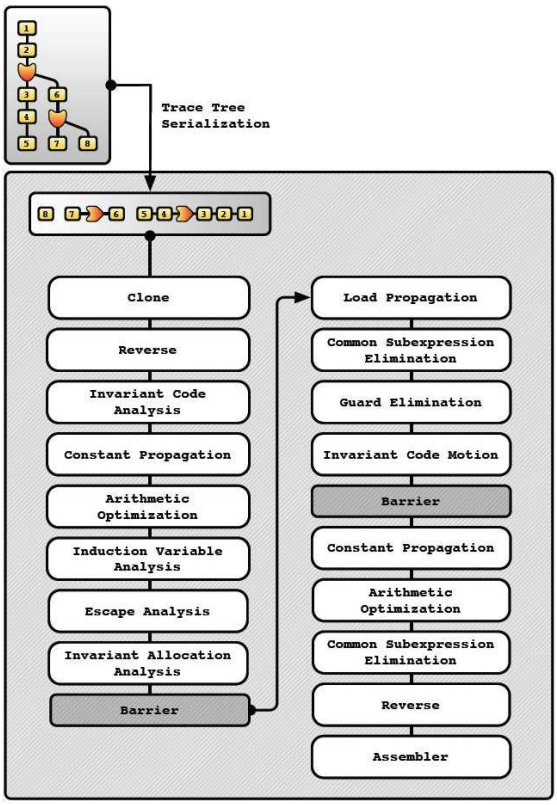

Figure 5: Hierarquia de classes do *pipeline*.

realizar a sincronização das *threads* que compõem o laço, consequentemente o *overhead* de sincronizar muitas *threads* pode superar os benefícios obtidos pela paralelização.

3. RESULTADOS DAS MODIFICAÇÕES

Os dados obtidos foram seperados em três partes: comparação da JamVM modificada com outras máquinas virtuais Java, comparação entre compiladores JIT sequencial e JIT paralelo, e compilador paralelizante dinâmico.

3.1 JamVM utilizanto Trace Tree

No primeiro artigo analisado[5], o autor modificou uma máquina virtual *open-source* chamada JamVM. O *software* compilado possue apenas 150kB, sendo que 2000 linhas de código correspondem apenas ao compilador JIT. Todos os programas foram comparados tomando como referência a *Sun Java Virtual Machine 1.5.0* com sua unidade JIT desabilitada. As outras máquinas virtuais testadas foram própria JVM 1.5.0, a SableVM 1.13 e a Cacao 0.96, sendo as duas últimas utilizadas em dispositivos móveis. O conjunto de programas foi criado pelo próprio autor[5][3] e executados em um computador PowerPC G5 1.86GHz com 1GB de memória RAM.

Analisando superficialmente (figura 7) vemos que a Cacao e a SableVM não apresentam ganhos médios significativos. Com ganhos médios de 7 e 10 vezes respectivamente, a JamVM é totalmente superada pela Sun JVM, mas esse ganho paga um preço inadimissível em dispositivos móveis e de recompilação dinâmica de código binário. Apesar de apresentar ganhos superiores, a Sun JVM consumiu 350 vezes mais

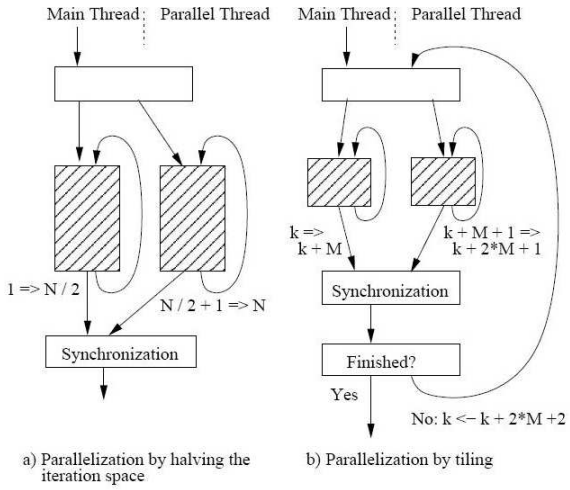

Figure 6: Exemplo de paralelização de laços.

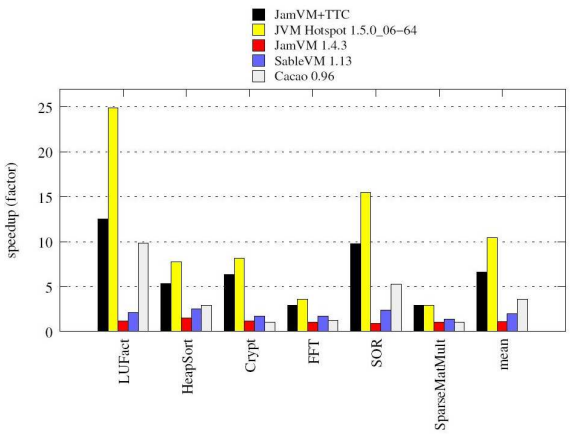

Figure 7: *Speedup* obtido pelo *Just-in-Time*.

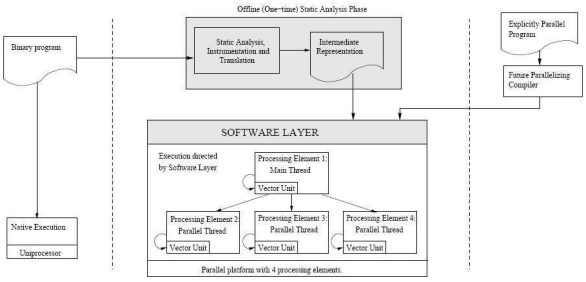

Figure 8: Estrutura do compilador dinâmico.

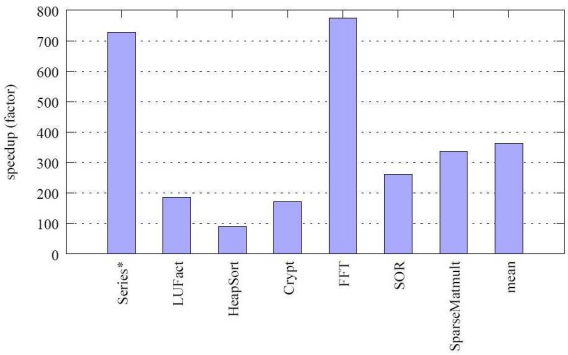

Figure 9: Tempo gasto pela JamVM comparada à Sun JVM.

tempo do que a JamVM. Além disso o compilador precisa armazenar em um *buffer* a representação intermediária do programa. Conseqüentemente a Sun JVM consome em média 7 vezes mais memória do que a JamVM, pois essa armazena apenas representações das partes significativas do programa. A utilização de *trace trees* na geração de código também diminui em 30 vezes a quantidade de código nativo emitido pela JamVM em comparativamente a Sun JVM. Esses dados podem ser visualizados nas figuras 9, 10 e 11.

3.2 Pipelined Trace Tree

O segundo conjunto de modificações[3] consiste na implentação em Java de duas versões do compilador em *pipeline*. A primeira consiste em realizar a compilação para código nativo seqüencialmente como uma etapa paralela a interpreteção. A segunda implementação executa cada etapa do *pipeline* separadamente em *threads*. Os testes foram realizados em um Apple MacBook Pro Core 2 Duo 2.16GHz com 2GB RAM.

A figura 12 mostra que a simples paralelização do compilador permite que os testes fossem executados 6 vezes mais rápido do que sua versão puramente seqüencial. Ainda mais interessante é constatar que a técnica é escalável, uma vez que o tempo necessário para a compilação cresce linearmente em relação a quantidade de instruções que compõem a árvore. Em compensação a versão *pipelined* não apresentou grandes diferenças devido a própria natureza das *trace trees*, pois não existe muito trabalho individual para cada etapa do processo. Além disso, executando apenas 4 *threads* simultaneamente, produzir 27000 instruções/s gerou um *overhead* de 108000 trocas de mensagens e a criação de 108000 temporários no *heap*.

3.3 Compilador Dinâmico

O terceiro artigo analisado apresentam execuções do *benchmarks* CPU SPEC2000 e Olden (treeadd em3d) compilados com o GCC 3.3. O *hardware* utilizado em multiprocessamento foi um PowerMac com dois processadores G4 866MHz. Já o *hardware* utilizado nos testes de *simultaneous multithreading* (SMT) e *chip multiprocessing* (CMP) foram executados em um Power5 1.67GHz. A tabela 1 mostra o número total de superblocos do programa, a quantidade de superblocos realmente instrumentados e percentual de laços identificados. A quantidade de paralelismo obtido depende diretamente da quantidade de laços paralelizáveis detectados.

A figura x mostra o ganho de desempenho nos testes do *benchmark* Olden. Quanto maior o número de instruções detectados, maior o *speedup* obtido. Isso ocorre porque *overhead* decorrente da criação de *threads* torna-se inexpressivo frente o ganho obtido pela paralelização dos programas. Desse modo o ganho limite proporcionado foi de 85%. A figura 13 mostra os *speedups* obtidos nas plataformas SMT e CMP. Ela mostra que o *overhead* da paralelização é quase imperceptível e um ganho médio de 10% para o *benchmark* de inteiros e 42% de *speedup* para o *benchmark* de ponto-flutuante.

4. CONCLUSÃO

A utilizaçã de *trace tree* em compiladores *Just-in-Time* permitiu criar um ambiente de compilação leve computacionalmente e capaz de oferecer um *speedup* semelhante a Sun JVM. Também é interessante perceber como esse processo de compilação é facilmente escalável e paralelizável em plataformas *multicore* se for programado como um *pipeline*.

O charme dessa estrutura esta no fato de ela ser suficientemente rápida para permitir a detecção dinâmica de laços de códigos binários executáveis sem prejudicar o desempenho do programa original. Como *trace trees* são superblocos, poderia ser implementada em um ambiente de recompilação dinâmica como o Dynamo[4] e combinado com as técnicas de paralelização e vetorização do terceiro artigo aqui estudado [9]. Esse novo tipo de compilador seria capaz de oferecer *speedups* consideráveis em aplicativos legados. Além disso, por gerar uma representação intermediária do programa, seria capaz de recompilá-lo para uma arquitetura diferente para a qual foi originalmente implementado.

Devemos salientar que os resultados obtidos pela análise estática seguida da paralelização dinâmica não utilizou otimizações de código clássicas, apenas paralelização e vetorização. Possivelmente seria capaz extrair restuldados ainda melhores se essas otimizações também fossem aplicadas.

5. REFERENCES

[1] A. Agarwal and M. Levy. The kill rule for multicore. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 750–753, New York, NY, USA, 2007. ACM.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers:*

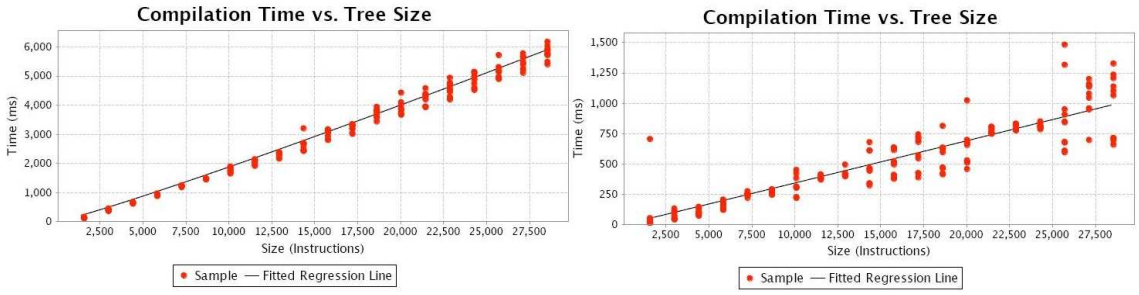

Figure 12: *Speedup* da compilação paralela a interpretação e da compilação em *pipeline*

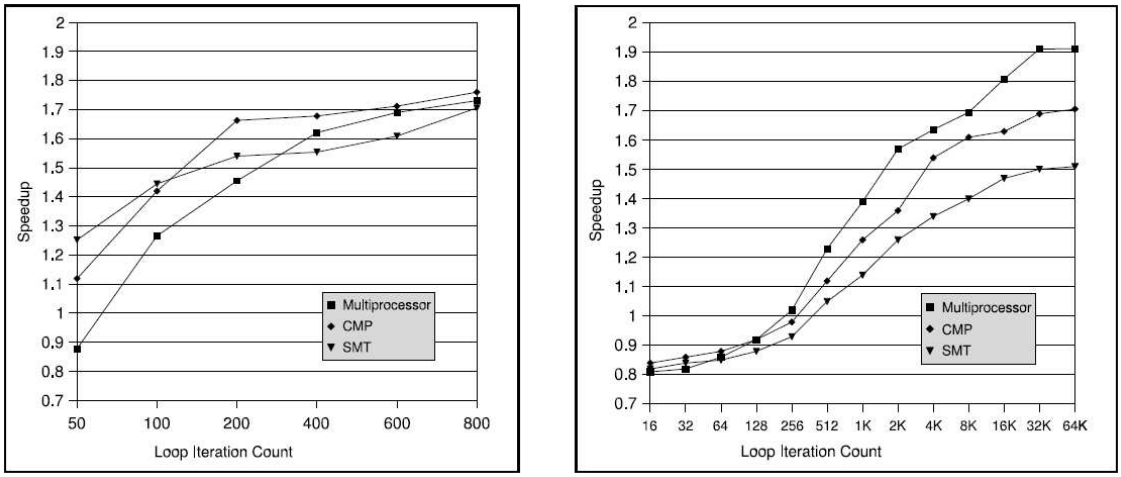

Figure 13: *Speedup* do Em3d e Treeadd (*benchmark* Olden.

Benchmark	gzip	vpr	mcf	parser	bzip2	twolf	perlbmk	art	swim	treeadd	em3d
Total number of superblocks	10900	12679	9415	13655	10839	13388	14736	12391	10012	9821	9318
Profi led superblocks	283	342	244	300	286	437	478	322	383	201	188
Access percentage of identifi ed loops	19.7%	0.1%	11.7%	5.8%	34.3%	31.0%	59.17%	58.72%	32.17%	97.8%	91.5%

Figure 14: Análise da quantidade de superblocos instrumentados.

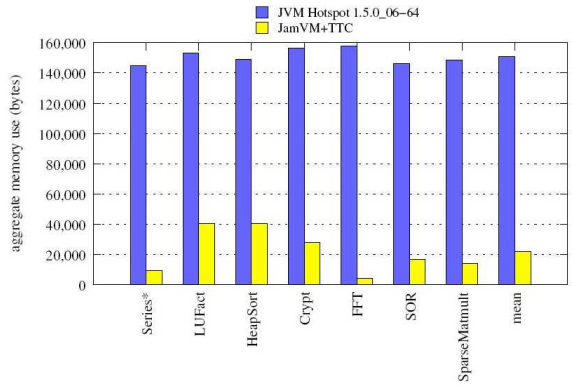

Figure 10: Consumo de memória do *Just-in-Time*.

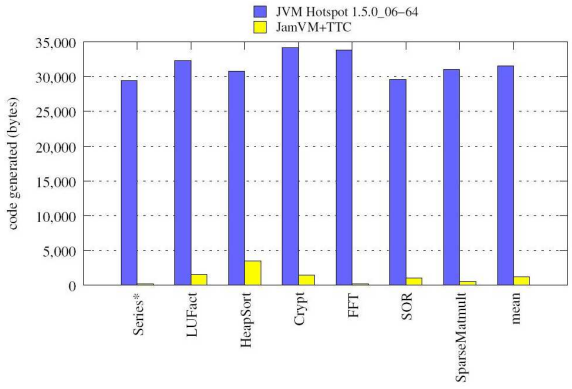

Figure 11: Quantidade de código nativo gerado.

principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] M. C. Andreas Gal, Michael Bebenita and M. Franz. Making the compilation of "pipeline" explicit: Dynamic compilation using trace tree serialization. In *Technical Report 07-12, Donald Bren School of Information and Computer Science*, pages 1–15, California, USA, 2007. University of California.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

[5] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. In *Technical Report 06-16, Donald Bren School of Information and Computer Science*, pages 1–11, California, USA, 2006. University of California.

[6] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach*. 2003.

[7] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. *SIGARCH Comput. Archit. News*, 20(2):46–57, 1992.

[8] D. Stefanovic and M. Martonosi. Limits and graph structure of available instruction-level parallelism (research note). In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 1018–1022, London, UK, 2000. Springer-Verlag.

[9] E. Yardimci and M. Franz. Dynamic parallelization and mapping of binary executables on hierarchical platforms. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 127–138, New York, NY, USA, 2006. ACM.