计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2020.20190524
57(2):243-256,2020

面向非易失内存写优化的重计算方法

张 铭 华 宇 刘璐荣 胡 蓉 李子怡
(武汉光电国家研究中心(华中科技大学) 武汉 430074)
(华中科技大学计算机学院 武汉 430074)
(csmzhang@hust.edu.cn)

A Write-Optimized Re-computation Scheme for Non-Volatile Memory

Zhang Ming, Hua Yu, Liu Lurong, Hu Rong, and Li Ziyi
(*Wuhan National Laboratory for Optoelectronics* (*Huazhong University of Science and Technology*), *Wuhan* 430074)
(*School of Computer*, *Huazhong University of Science and Technology*, *Wuhan* 430074)

Abstract The rise of non-volatile memory (NVM) technology brings many opportunities and challenges to computer storage systems. Compared with DRAM, NVM has the advantages of non-volatility, low energy consumption and high storage density as persistent memory. However, it has the disadvantages of limited erase/write times and high write latency. Therefore, it is necessary to reduce the write operations to the non-volatile main memory to improve system performance and extend the lifetime of NVM. To address this problem, this paper proposes ROD, which is a re-computation scheme based on the out degree of computing nodes. Due to the performance gap between CPU and memory leading to computing resources waste, ROD selectively discards the computed results which should be stored into the NVM and re-computes them when needed. By swapping the storage with computation, ROD reduces the number of writes to the NVM. We have implemented ROD and evaluated it on Gem5 simulator with NVMain. We also implemented the greedy re-computation scheme and the store-only scheme as state-of-the-art work to be compared with ROD. Experimental results from powerstone benchmark show that ROD reduces write operations by 44.3% on average (up to 68.5%) compared with the store-only scheme. The execution time is reduced by 28.1% on average (up to 68.6%) compared with the store-only scheme, and 9.3% on average (up to 19.4%) compared with the greedy scheme.

Key words non-volatile memory (NVM); re-computation; write reduction; node's out degree; simulator

摘 要 非易失存储(non-volatile memory, NVM)技术的兴起给计算机存储系统带来了很多机遇与挑战.与DRAM相比,NVM作为持久性内存具有非易失、低能耗以及高存储密度等优点,但同时它也具有擦/写次数有限以及写操作延迟高等缺点,故以NVM为内存的系统需要减少对内存的写操作,以提升NVM寿命和系统性能.为了解决这个问题,提出了基于结点出度的重计算方法(re-computation scheme based on the out degree of computing nodes, ROD),由于CPU与内存间的性能差距会导致CPU计算资源的浪费,为此ROD方法选择性地丢弃本需要存储到内存的计算结果,需要时再重新计算得到,利用计算换存储的方式减少写NVM的次数.实验采用powerstone测试集,在搭载了NVMain的

收稿日期:2019-08-12;**修回日期**:2019-11-18
基金项目:国家自然科学基金项目(61772212)
This work was supported by the National Natural Science Foundation of China (61772212).
通信作者:华宇(csyhua@hust.edu.cn)

Gem5 模拟器中对 ROD 方法与贪心重计算方法和以存储为主导的无重计算方法做性能对比.结果表明 ROD 方法相比于存储主导的方法平均减少 44.3%(最高 68.5%)的写操作.ROD 方法的运行耗时比存储主导的方法平均减少 28.1%(最高 68.6%),比贪心重计算的方法平均减少 9.3%(最高 19.4%).

关键词 非易失存储;重计算;减少写;结点出度;模拟器

中图法分类号 TP302

随着大数据时代数据规模的快速增长,数据存储与访问给内存带来了更高的性能要求.传统的 DRAM 技术因其易失性以及刷新功耗大等问题在系统的可靠性以及能耗等方面会面临诸多挑战.而非易失存储器[1](non-volatile memory, NVM)可以按字节寻址用作持久性内存[2],具有非易失、能耗低以及存储密度高等优点,为构建更高效的存储系统带来了机遇,是下一代内存的理想选择.按字节寻址的非易失存储器主要包括相变存储器[3-4](phase change memory, PCM)、阻变存储器[5](resistive random access memory, RRAM)、磁随机存储器[6](magnetic random access memory, MRAM)以及自旋矩存储器[7](spin transfer torque random access memory, TT-RAM)等.英特尔公司与镁光科技公司于 2015 年公布了非易失存储技术 3D XPoint[8],并于 2018 年发布了基于 3D XPoint 的商用非易失 DDR4 内存 Optane DC Persistent Memory[9],可以配合现有的 Skylake-SP 架构 Intel Xeon 处理器用于数据服务器.随着硬件技术日渐成熟,基于 NVM 的下一代存储系统也成为了当前的研究热点[10].

然而使用 NVM 作为内存应用于计算机系统中仍面临着诸多挑战,这主要包括 2 点:一是 NVM 的写延迟比 DRAM 高[11];二是 NVM 的擦/写次数有限,使用寿命比 DRAM 短[12].因此如何优化 NVM 的写操作是一个非常重要的问题.

现有的解决方案从写暂停、对比写、翻转写、减少写数据量以及磨损均衡[13]等角度对 NVM 的写操作做了优化.由于 NVM 读操作的延迟和耗能都低于写操作,因此 Qureshi 等人[14]提出了写取消及写暂停策略,使得系统可以优先处理读请求以缩短系统响应速度.Yang 等人[15]利用 NVM 的字节寻址特性提出了对比写的策略,在数据更新时通过对比新旧数据只修改发生变化的比特.Cho 等人[16]提出了翻转写的策略,使用一个标志位记录翻转操作,当修改的数据位超过一半时,翻转标志位和未修改的数据位,当修改的数据位不超过一半时直接写入修改的数据位并保持标志位不变,以此保证数据更新时修改的数据位不超过一半.Tseng 等人[17]使用线性规划方法寻求任务的最佳调度顺序以减少对脏数据的写回,从而达到减少写数据量的目的.Chen 等人[18]使用计数器以及基于桶的磨损均衡算法为物理页维护一个空闲页面和磨损避免页面以达到磨损均衡的目的,从而提高 NVM 的耐久性和使用寿命.

为了解决 NVM 写延迟高和写寿命短的问题,与上述解决方案不同,本文从计算替换存储的角度提出了一种基于结点出度的重计算方法称作 ROD(re-computation scheme based on the out degree of computing nodes),ROD 方法利用 NVM 材料读写延迟的不对称性,通过读取输入数据重新计算代码块的结果以减少对 NVM 的写次数.具体而言,首先按照程序指令间的数据依赖关系在编译期构造数据流图[19](data flow graph, DFG),DFG 中的每个结点表示一条程序语句,从输入开始到输出结束,再跟据指令执行周期以及 NVM 的读写延迟对 DFG 决策出所有的存储结点和重计算结点,最后根据重计算结点的计算路径完成重计算过程.其中存储结点表示该结点的计算结果写入 NVM,需要使用时直接从内存中读出,重计算结点表示该结点的计算结果不写入 NVM,当使用时需要回溯到最近的存储结点并将计算路径重新执行一遍.传统的存储主导的方法便是将所有计算结果存入内存,需要使用时再从内存中取出.一种贪心重计算的方法是只对当前结点的重计算开销和存储开销做比对从而作出决策.由于在 DFG 中结点的重计算路径越长则开销越大,因此需要适当地选择存储结点,而结点的出度决定了数据被依赖的程度,出度越高说明结点被使用的次数越多,读开销或重计算开销也就越大,这是贪心方法未考虑到的,因此 ROD 方法结合结点的出度可以做准确的决策.

本文的主要贡献包括 3 个方面:

1) 提出了基于结点出度的重计算方法称作 ROD,从而减少 NVM 的写操作,缓解 CPU 与内存之间的 I/O 开销.

2) 设计并实现 ROD 方法的 3 个组成模块,分别是数据流图的构建、结点的决策算法以及重计算路径的生成.

3) 使用搭载 NVMain 的 Gem5 模拟器对 ROD 方法、贪心重计算方法以及存储主导的方法做了性能对比,验证了 ROD 方法的有效性.

1　研究背景

本节首先介绍现代 CPU 与内存之间的性能差距,这是重计算的动机所在,然后介绍重新计算的定义和内涵,接着描述重计算路径图,最后介绍对程序语句执行结果的重计算或存储的权衡策略.

1.1　CPU 与内存间性能瓶颈

现代 CPU 性能比内存性能高 3 个数量级[20],频繁读写内存不仅会缩短 NVM 的使用寿命,而且较大的 I/O 延迟会导致 CPU 空转,从而浪费计算资源并降低系统性能.

在传统的执行方式中,程序的中间计算结果写入内存,需要时再从内存中读出.考虑到 CPU 与内存间的性能瓶颈,重计算方法通过丢弃部分需要写入内存的计算结果,需要时再按计算路径重新计算出来,以此降低 I/O 的开销,对于以 NVM 为内存的系统即可减少对内存的写操作,提高 NVM 的使用寿命.

1.2　定义及研究方向

与重计算相关的研究主要可分为 2 类:1)降低系统访存次数;2)系统容错和恢复.

对于第 1 类研究,重新计算是指在需要程序的中间计算结果时将其重新计算出来,以替代将计算结果写入内存并在需要时读取出来的方案[21].该方案的优势在于可以减少写 NVM 的次数,提高 NVM 的耐久性并降低 I/O 开销,但问题在于会带来额外的计算开销.因此需要合理地对计算结果做存储或重计算的决策,以较少的额外计算开销换取较大的写开销.

对于第 2 类研究,重计算大多应用于高性能计算系统的容错和恢复中[22],在应用程序运行崩溃时,保存在 NVM 上的数据不会丢失,即可恢复计算场景以重新计算出结果,因此如何减少保存的数据量和加快系统恢复速度是其主要的研究问题.

这 2 类研究方向的具体工作将在第 4 节"相关研究工作"做详细介绍.本文的研究内容是利用重计算降低系统访存次数,以减少对 NVM 的写操作.

1.3　重计算路径

由于重计算不保存部分计算结果,因此需要确定重新计算执行的指令及其顺序,来保证程序运行的正确性.将程序按数据依赖划分为代码段,为了方便描述,使用生产者与消费者的概念,如图 1 所示,代码段 G 需要直接使用代码段 F 的运行结果,称 F 是 G 的直接生产者,也即 G 是 F 的直接消费者.若代码段 F 的计算结果没有保存至 NVM,那么在执行代码段 G 时需要重新运行代码段 F 以产生 G 的输入数据.若 F 的直接生产者 D 的计算结果没有保存,那么需要递归回溯到最近的保存了计算结果的生产者.由此可知,将耗能较高的读写内存操作替换为计算指令,便可得到重计算路径,它是一条沿着数据流递归回溯的路径.

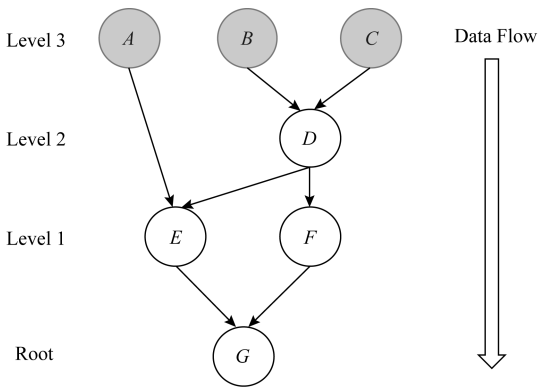


Fig. 1　The re-computation path

图 1　重计算路径图

图 1 展示了一个具有 4 层结点的重计算路径图,这是一棵倒长的树.其中 A,B,C 表示程序的输入,其结果需要写入内存;D,E,F 为中间计算结点,不保存它们的计算结果;G 为程序输出结点.第 1 层的结点对应于根结点的直接生产者,从第 2 层往上,第 i 层的结点对应于第 $i+1$ 层结点的直接消费者.每个结点的入度表示其直接生产者的个数,出度表示其直接消费者的个数.由于代码段 D,E,F 的计算结果没有保存,因此在计算 G 时,需要重新计算 D,E,F 的结果,故需要沿着数据流回溯到结点 A,B,C,从内存中读取数据重新计算.

1.4　权衡策略

重计算虽然能减少对 NVM 写的次数,但如果对于代码段执行结果的存储或重计算的决策不佳,可能会使得重计算路径过长导致计算开销过大,或者存储结点过多导致写 NVM 开销增大.本节介绍对代码段执行结果的存储或重计算的权衡策略.

考虑图 2 所示的 2 种重计算路径形态,图 2 中灰色圆形表示程序的输入,其结果保存到内存,结点 G 表示程序的输出.图 2(a)的结点层数较多,只保存输入结点,会使得后续结点的重计算路径过长,需要递归回溯执行的指令过多,更优的策略应适当存储中间的计算结果,避免重计算开销过大.而图 2(b)的结点层数较少,计算路径短,将中间计算结果都存入 NVM 必然会造成大量的写开销,因此对中间结点做重计算的优势能更好地体现出来.

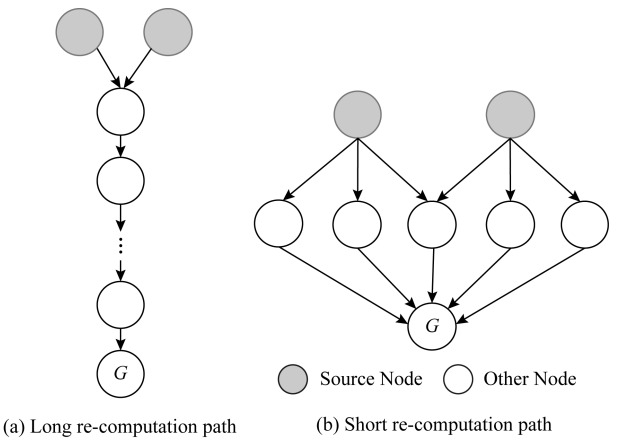


Fig. 2 Different forms of re-computation path

图 2 不同形态的重计算路径

Kandemir 等人[23]提出了贪心的决策方法,即只考虑当前的计算结果,当保存到内存的开销小于将其重新计算出来的开销,即使用存储策略,否则使用重计算策略.但是贪心方法忽略了数据的使用次数,当一个数据在短期内被重复使用时,选择保存该数据的结果能获得更大的收益.

2 基于结点出度的重计算方法

为了减少对 NVM 的写操作,提出基于结点出度的重计算方法 ROD,其考虑指令间数据依赖的程度,对计算结果做出合理的存储或重计算的决策.本节先介绍 ROD 方法的总体设计,再介绍各个组成模块的具体实现.

2.1 总体设计

本节介绍 ROD 方法的总体流程设计,共有 3 个主要的功能模块,分别是数据流图的生成模块、结点决策模块以及重计算路径的生成模块,如图 3 所示.

ROD 方法通过数据流对程序按语句划分结点,一条语句即为一个结点,结点之间的箭头指向关系即表示生产者结点与消费者结点之间存在数据依

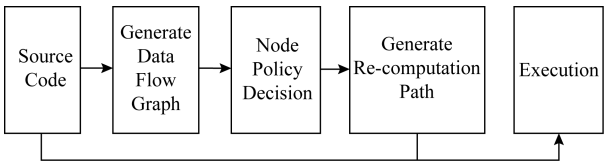


Fig. 3 The process design of ROD scheme

图 3 ROD 方法总体流程设计

赖.在本方法中,数据分为可重计算数据与非可重计算数据,程序的输入数据属于非可重计算数据,除了输入结点,中间结点的重计算开销大于存储开销的数据也属于非可重计算数据,其他数据则属于可重计算数据.产生非可重计算数据的结点称为存储结点,其计算结果需写入到 NVM 中.产生可重计算数据的结点称为重计算结点,其计算结果被使用到时需按照数据依赖关系回溯到最近的存储结点,将其值读出后按计算路径重新计算出结果.

为了得到结点之间的数据依赖关系必须先构造数据流图.使用 LLVM[24]工具对源程序生成中间表示(intermediate representation, IR)指令,即可根据 IR 表示中的 load/store 指令找到对应的结点,再使用词法分析的方式对这些结点提取对应的操作数,最后依赖操作数之间的读写依赖构造数据流图,如图 4 所示,图中每个结点表示一条语句,结点 V_i 指向结点 V_j 表示第 j 条语句的执行依赖第 i 条语句的计算结果.

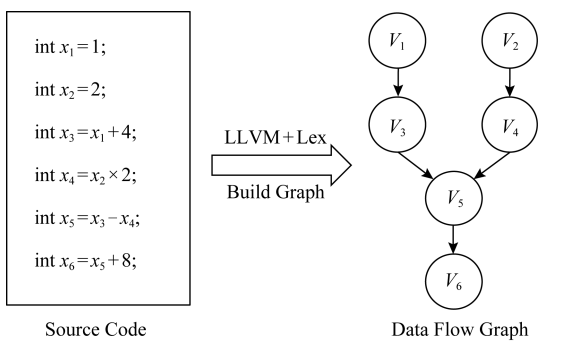


Fig. 4 Transform the source code to the data flow graph

图 4 将源程序转化为数据流图

在得到数据流图之后,便可以计算结点的存储开销和重计算开销来确定存储结点和重计算结点.ROD 方法在决策的过程中综合考虑了结点出度(即直接消费者个数)的影响,结点的出度越大,说明其所在的重计算路径越多,因此保存其计算结果往往会有利于其他结点的重计算过程.

例如,经过决策后将图 4 的数据流图转换为决策图,其中 S 结点表示存储结点,R 结点表示重计算结点,如图 5 所示,可以看出程序的输入和输出结点都被标为存储结点,其他中间计算结果均可通过重新计算得到,比如计算结点 V_5 需要重新计算结点 V_3 和结点 V_4,而计算结点 V_6 则需要重新计算结点 V_3、结点 V_4 和结点 V_5.

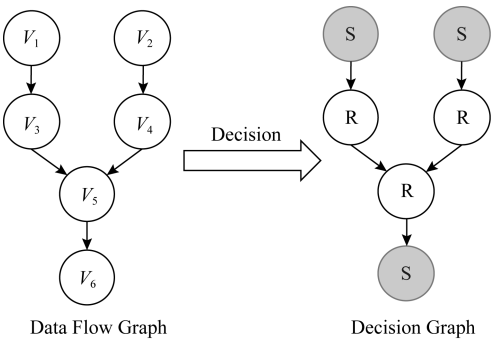


Fig. 5　The decision of the store node and the re-computation node

图 5　存储结点和重计算结点的决策

通过决策图 5 可以回溯每个重计算结点的重计算路径,重计算路径应从重计算结点开始一直回溯到距离它最近的存储结点为止,以此为每个重计算任务生成重计算函数.由于本方法是在编译期确定重计算结点,在所有重计算函数确定后,便可以运行程序,在遇到重计算结点时进入相应的重计算函数得到计算结果即可,因此程序实际执行的指令数量也相应地增多了.

2.2 节将详细介绍数据流图生成的实现,2.3 节将详细介绍 ROD 方法中对于存储或重计算选择策略的实现细节,2.4 节将对如何寻找重计算路径并生成重计算函数做详细的介绍.

2.2　生成数据流图

本节将介绍生成数据流图的实现细节.通过使用 LLVM 工具链中 Clang 编译器为源程序生成 IR 指令,通过分析 IR 中的 load/store 指令在源程序中找到对应的读写内存的语句,其中 load/store 指令的具体含义表示如下:

1) load 指令.读内存,需要记录读取的地址与存入的操作数.

2) store 指令.写内存,需要记录被保存的数据与写入的地址.

对含 load 和 store 操作的 IR 指令举例如下:

$\%p$=alloca i32;

store i32 8,i32 * $\%p$;

$\%v$=load i32,i32 * $\%p$;

首先向内存申请一个 32 b 的整型空间 p,然后将 32 b 的数 8 写入地址 p 处,再从内存中读出地址 p 处的值,将其存入 v 中,此时 v 的值为 8.

在得到所有读写内存的程序语句之后便可以利用词法分析的方式提取左右操作数存入每个结点中.在这之后便可对每条语句的左操作数遍历匹配所有在其之后语句的右操作数,匹配成功就说明语句间有数据读写关系,将二者连接起来便构成数据流图的边.这个过程中要考虑到操作数被覆盖写(overwrite)的情况,即结点 A 和结点 B 含有相同作用域的左操作数 L_{op},且 A 的语句顺序在 B 之前,那么在遍历到 B 时,对 A 的 L_{op} 的匹配任务应立即停止,因为 L_{op} 将被 B 的计算结果写覆盖,B 之后所有用到 L_{op} 的结点将不再依赖 A.在得到所有的结点和边之后,便可通过构造邻接表(adjacency list)的方式构造数据流图,用于决定结点的存储或重计算的策略.

2.3　存储与重计算策略

介绍 ROD 方法所使用的结点存储或重计算决策模型的设计与实现细节.使用 V_i 表示每个结点,定义写内存开销为 $M_w(V_i)$,读内存开销为 $M_r(V_i)$.使用 0-1 变量 α_i 来决定是否将结点 V_i 的计算结果存入内存,如式(1)所示:

$$\alpha_i=\begin{cases}1, & \text{保存计算结果;}\\ 0, & \text{不保存计算结果.}\end{cases} \tag{1}$$

定义每个结点的生产开销为 $P(V_i)$,如式(2)所示,分为计算开销 $C(V_i)$ 和保存计算结果的开销 $M_w(V_i)$ 这 2 部分.

$$P(V_i)=C(V_i)+\alpha_i M_w(V_i), \tag{2}$$

其中计算开销 $C(V_i)$ 的定义如式(3)所示,包括结点计算自身结果所产生的开销 $o(V_i)$ 以及访问其所有直接生产者的开销 $D(V_j)$.

$$C(V_i)=\sum_{V_j\in preV_i}D(V_j)+o(V_i), \tag{3}$$

结点数据要么从内存中读出,要么通过计算产生,因此结点的直接生产者开销 $D(V_j)$ 如式(4)所示:

$$D(V_j)=\alpha_j M_r(V_j)+(1-\alpha_j)C(V_j), \tag{4}$$

当 $\alpha_j=1$ 时,说明其生产者结点为存储结点,只存在读内存的开销 $M_r(V_j)$,当 $\alpha_j=0$ 时,说明其生产者结点为重计算结点,因此需要使用式(3)递归计算其开销.由此可以看出重计算路径越长,其递归层数越深,重计算开销也就越大.

1.4 节介绍了一种贪心的决策方法,即只根据当前结点的重计算开销和存储开销做决策,不考虑所做的决策对其消费者产生的读开销,即:

1) 若当前结点 V_i 采用存储策略,则将计算结果保存到内存,需要用到时直接从内存中读取,产生的开销为 $M_w(V_i)$;

2) 若当前结点 V_i 采用重计算策略,则不保存计算结果,需要时重新计算产生,产生的开销为 $C(V_i)$.

由于程序的输入结点不可重计算,因此输入结点均采用存储的方案,其余结点则需要比较 $M_w(V_i)$ 与 $C(V_i)$ 之间的大小关系,$M_w(V_i)<C(V_i)$ 即采用存储策略,否则采用重计算策略.贪心重计算方法在比较存储开销和重计算开销时并没有考虑到当前结点的直接消费者个数对于决策的影响.通常直接消费者结点个数即出度越高的结点,其计算结果被用到的次数也越多,因此选择性地将其计算结果存储下来有利于其消费者结点的重计算过程.

为此提出一种新型的决策模型.记每个结点 V_i 的出度为 φ_i,那么该结点若使用存储策略,则其计算结果将被读 φ_i 次,若使用重计算策略,则需要被重计算 φ_i 次.因此结点的存储开销 W_S 和重计算开销 W_R 如式(5)所示:

$$\begin{aligned} W_S(V_i)&=M_w(V_i)+\phi_i M_r(V_i),\\ W_R(V_i)&=\phi_i C(V_i), \end{aligned} \tag{5}$$

其中 $C(V_i)$ 的定义同式(3),当 $W_S<W_R$ 时将结点的计算结果写入内存中,否则记为重计算结点.根据上述决策方案的形式化表达,ROD 方法对结点决策的算法伪代码如下:

算法 1. 结点存储或重计算的决策算法.

输入:程序的数据流图 DFG;

输出:存储结点集合 A 和重计算结点集合 B.

① $A\leftarrow\varnothing$, $B\leftarrow\varnothing$;
② for each node N in DFG do
③ if N 是输入结点 then
④ add N to set A;
⑤ else
⑥ $P\leftarrow$producer of N;
⑦ $\varphi\leftarrow$the out degree of N;
⑧ $RCost\leftarrow 0$; /* $RCost$ 为重计算开销 */
⑨ while P is not NULL do
⑩ add P's cost to $RCost$;
⑪ $P\leftarrow N$'s next producer;
⑫ end while
⑬ $RCost\leftarrow\varphi\times(RCost+o(N))$;
⑭ $SCost\leftarrow\varphi\times RDTIME+WRTIME$;
⑮ if $SCost<RCost$ then
⑯ add N to A;
⑰ else
⑱ add N to B;
⑲ end if
⑳ end if
㉑ end for
㉒ 输出集合 A 和 B.

算法 1 的行③~⑤表示存储所有的输入结点,第⑥行的 P 表示结点 N 的一个生产者,由于采用邻接表法表示数据流图,因此结点 N 的其他生产者均可以由 P 访问到.行⑨~⑫表示累加结点 N 的所有生产者的开销,得到 $\sum_{P\in preND(P)}$,再加上结点 N 自身的计算开销 $o(N)$,最后利用式(3)(5)即可得到结点 N 的重计算开销 $RCost$.在决策时认为每条 IR 指令运行的 CPU 时钟周期数为 1,通过 IR 指令的数量与 C 程序语句的数量之比即可确定计算每个结点的时钟周期数.行⑭通过式(5)计算结点 N 的存储开销,在实现中采用的是 PCM 的读写延迟.行⑮~⑲通过比较 $RCost$ 与 $SCost$ 来决定结点 N 的计算结果是存储还是重计算.

为了更好地理解重计算过程,对不同的存储与重计算策略进行举例说明.图 6 是包含 7 个结点的数据流图,共有 3 个输入结点、1 个输出结点以及 3 个中间结点,其中图 6(a)表示只存储输入结点的数据,记为决策(a).图 6(b)表示不仅存储输入结点的数据,还存储中间结点 V_4 的计算结果,记为决策(b).

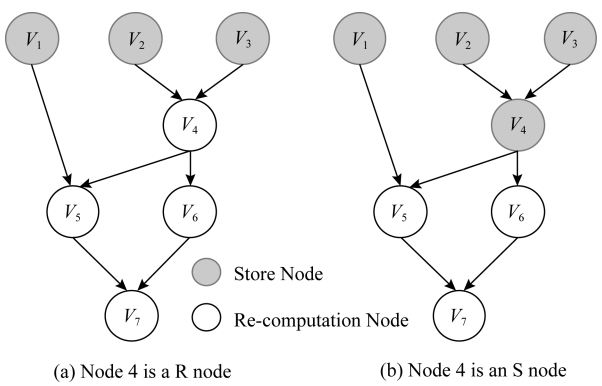


Fig. 6 Different decisions on store or re-computation

图 6 关于结点存储或重计算的不同决策

在传统的执行流程中需要将 7 个结点的计算结果都写入 NVM,但决策(a)中只存储结点 $V_1\sim V_3$,决策(b)中只存储结点 $V_1\sim V_4$.决策(a)在执行过程中对每个结点所产生的操作如表 1 所示:

Table 1　Operations on Each Node of Decision (a)

表 1　决策(a)对每个结点的操作

Node	Operations
V_1	Input,$M_w(V_1)$
V_2	Input,$M_w(V_2)$
V_3	Input,$M_w(V_3)$
V_4	$M_r(V_2)$,$M_r(V_3)$,$o(V_4)$
V_5	$M_r(V_1)$,$M_r(V_2)$,$M_r(V_3)$,re-compute(V_4),$o(V_5)$
V_6	$M_r(V_2)$,$M_r(V_3)$,re-compute(V_4),$o(V_6)$
V_7	$M_r(V_1)$,$M_r(V_2)$,$M_r(V_3)$,re-compute($V_{4,5,6}$),$o(V_7)$

从表 1 可以看到决策(a)产生了 10 次读 NVM、3 次写 NVM、5 次重计算,相比较于传统执行流程减少了 4 次写 NVM 的操作.决策(b)在执行过程中对每个结点所产生的操作如表 2 所示:

Table 2　Operations on Each Node of Decision (b)

表 2　决策(b)对每个结点的操作

Node	Operations
V_1	Input,$M_w(V_1)$
V_2	Input,$M_w(V_2)$
V_3	Input,$M_w(V_3)$
V_4	$M_r(V_2)$,$M_r(V_3)$,$o(V_4)$,$M_w(V_4)$
V_5	$M_r(V_1)$,$M_r(V_4)$,$o(V_5)$
V_6	$M_r(V_4)$,$o(V_6)$
V_7	$M_r(V_1)$,$M_r(V_4)$,re-compute($V_{5,6}$),$o(V_7)$

从表 2 可以看到决策(b)产生了 7 次读 NVM、4 次写 NVM、2 次重计算,相比较于传统执行流程减少了 3 次写 NVM 的操作.

这 2 种决策方案相较于传统的执行流程都可以减少对 NVM 的写操作,虽然决策(a)比决策(b)少了 1 次写操作,但却多出了 3 次读操作开销和 2 次重计算开销.由此可以看出不同的决策方案即使相似,也可以对系统性能产生截然不同的影响.

2.4　重计算路径的生成

通过 2.3 节的结点决策算法得到决策图后,下一步便是从决策图中寻找重计算路径,对此部分做详细介绍.

由于重计算结点并不保存计算结果,因此需要从其生产者结点开始回溯到最近的存储结点,从 NVM 中读取相应的值,再从存储结点开始顺着回溯的路径重新计算出所需的结果,这条回溯的路径便是重计算路径,本文通过构造重计算函数来实现重计算路径.

第 1 步.为了找到回溯的路径,需要记录每个重计算结点的直接生产者,通过逐级查找的方式找路径.为了达到这个目的,需要在构造数据流图时通过结点链表记录每个结点指向的所有生产者结点,在经过 ROD 方法的决策之后,便可以得到所有的重计算结点及其直接生产者结点的关系表,进而找到重计算路径.例如,对于图 4 的源代码及图 5 的决策结果,其重计算结点及重计算路径如表 3 所示:

Table 3　Re-computation Nodes and Their Re-computation Paths

表 3　重计算结点及其重计算路径

Re-computation Nodes	Direct Producers	Re-computation Paths
V_3	V_1	$V_3 \rightarrow V_1$
V_4	V_2	$V_4 \rightarrow V_2$
V_5	V_3,V_4	$V_5 \rightarrow V_3 \rightarrow V_1$ $V_5 \rightarrow V_4 \rightarrow V_2$

ROD 方法采用深度优先全路径遍历的方式寻找重计算路径.对于结点 V_3,其直接生产者为 V_1,但 V_1 是存储结点,没有直接生产者,因此重计算 V_3 的路径为 $V_3 \rightarrow V_1$.同理可得结点 V_4 的重计算路径为 $V_4 \rightarrow V_2$,对于结点 V_5,其直接生产者 V_3 和 V_4 均可在表 3 中找到,需要对 V_3 和 V_4 做递归遍历,因此其路径遍历的结果为 $V_5 \rightarrow V_3 \rightarrow V_1$ 和 $V_5 \rightarrow V_4 \rightarrow V_2$.

第 2 步.在找到每个重计算结点的重计算路径之后,需要实现重计算这个过程,因此需要构造重计算函数,构造重计算函数的过程实质上就是根据重计算路径做递归计算的过程.例如,对于上述的重计算结点 V_3,V_4,V_5,它们的重计算函数 rec_x_3,rec_x_4,rec_x_5 如图 7 所示:

```
int rec_x3(int x1){
    return x1+4;
}
int rec_x4(int x2){
    return x2×2;
}
int rec_x5(int x1,int x2){
    return rec_x3(x1)−rec_x4(x2);
}
int main(){
    int x1=1;
    int x2=2;
    int x3=x1+4;
    int x4=x2×2;
    int x5=rec_x3(x1)−rec_x4(x2);
    int x6=rec_x5(x1,x2)+8;
}
```

Fig. 7　The implementation of re-computation functions

图 7　重计算函数的实现

从图 7 可以看出,每个重计算结点都有对应的重计算函数,并且随着数据依赖的加深,其重计算需要递归的层数也在加大,这将使程序的指令数增大,同时也加大了重计算的开销.

重计算方法虽然具有减少写 NVM 次数的优点,但也会有计算开销大的缺点,因此在实际的决策过程中,要充分考虑到 NVM 材料的读写开销与指令的执行开销,选择保存出度较大的结点的计算结果可以对后续结点的重计算过程更有利,这也是 ROD 方法的核心决策观点.

3 实验结果与分析

在实现数据流图的生成、结点决策算法以及重计算路径的生成等模块之后,使用 Gem5 模拟器对比了 ROD 方法、存储主导方法和贪心方法的性能.

3.1 实验环境

在 Ubuntu 操作系统上使用 LLVM 套件,C++语言以及 g++编译器完成各功能模块的开发部分,各开发工具的版本号如表 4 所示:

Table 4 The Version Configurations of Development Tools

表 4 开发工具的版本配置

Develop Tools	Versions
Ubuntu Operating System	14.04 LTS
LLVM	8.0.0
C++	Standard 11
G++Compiler	4.8.4

完成开发后使用 Gem5[25]模拟器对各方法做性能评测.Gem5 模拟器用于计算机系统架构的相关研究,包括系统级架构和处理器微架构.为了模拟非易失内存的读写延迟,使用 NVMain[26]作为插件配合 Gem5 做模拟,NVMain 是一个体系结构级的非易失内存模拟器,可以准确地模拟内存系统的时序和能耗.Gem5 实现了 x86 指令集中的 clflush[27]指令和 mfence[28]指令,可以利用这些指令将缓存行的数据有序的刷回 NVM 中,Gem5 模拟的系统配置如表 5 所示,其中 NVMain 的配置与 Choi 等人[29]提出的 PCM 配置相同,使用 NVMain 内置的 PCM_ISSCC_2012_4GB.config 即可,其中 PCM 的频率为 500 MHz,读延迟为 120 ns,写延迟为 150 ns.

实验采用的测试集为 powerstone benchmark[30],它包含一系列嵌入式和可移植的应用程序,包括分页、自动控制、信号处理以及图像处理等方面,程序中数据依赖关系明显,具有不同的访存特征.实验选取了 powerstone benchmark 部分测试程序,各测试程序的描述[30]如表 6 所示.

Table 5 The Configurations Parameters of Gem5

表 5 Gem5 配置参数

Configuration Items	Configuration Parameters
Operating System	Ubuntu 14.04 LTS
System Mode	SE
CPU Type	Timing
CPU Clock/GHz	2.3
L_1 Data Cache Size/KB	64
L_1 Instruction Cache Size/KB	16
L_2 Cache Size/MB	2
Cache Line Size/B	64

Table 6 The Description of Benchmark

表 6 Benchmark 各程序描述

Benchmark	Description
qurt	Square root calculation using floating point
g3fax	Group three fax decode (single level image decompression)
fir	Integer FIR filter
engine	Engine control application
crc	Cyclic redundancy check
blit	Graphics application
bcnt	Bit shifting & anding through 1K array

为验证 ROD 方法的有效性,对比了传统的无重计算的存储主导方法和 1.4 节所描述的贪心方法,并测试在不同 PCM 读写延迟下 3 种方法的运行耗时.

3.2 实验结果

1) 测试不同程序按数据依赖划分出的结点数,包括输入结点数和非输入结点数.实验中提取了每个测试程序的所有赋值语句,每条赋值语句为一个结点,赋值语句之间存在典型的数据依赖关系.赋值语句中不可被重计算的属于输入结点,其他可以被重计算的属于非输入结点.测试结果如图 8 所示.

分析上述实验结果,除了 g3fax 和 qurt,不同应用程序的非输入结点数较多,平均占比约为 65%,由于只有非输入结点才可以被重计算,因此程序的输入结点越少,临时的存储开销就越少,重计算方法的多算少读的优势就会越明显.

2) 测试 ROD 方法对结点的决策结果.由于程序输出结点的计算结果一般会被写入 NVM,因此

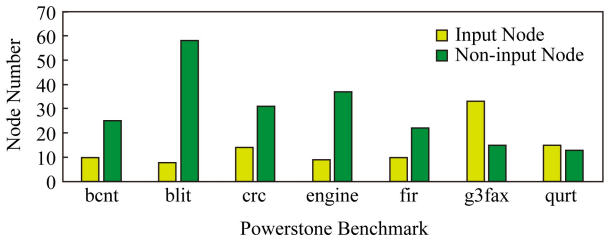


Fig. 8　The number of input nodes and non-input nodes of different programs

图 8　不同测试程序的输入结点数和非输入结点数

每个程序的存储结点数会比输入结点数多.ROD 方法通过结点的出度权衡读开销和重计算开销,其决策结果如图 9 所示:

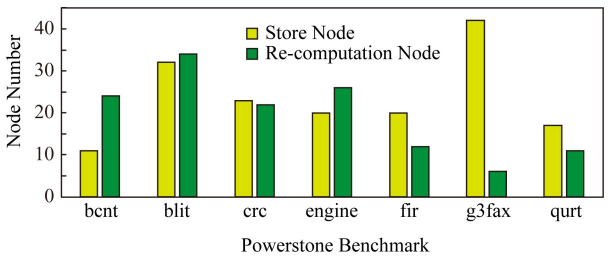


Fig. 9　The decision results of ROD scheme

图 9　ROD 方法的决策结果

分析上述实验结果,g3fax 的重计算结点数较少,占比总结点数的 12.5%,所有程序的重计算结点数占比平均为 44.3%,最高为 68.5%.这也说明了使用 ROD 方法最多能减少程序 68.5%的写 NVM 操作,这一方面可以减少 I/O 带来的延迟,另一方面也能提高 NVM 的寿命.

3) 测试贪心方法对结点的决策结果,结果如图 10 所示:

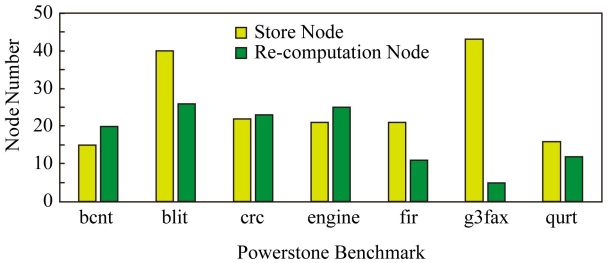
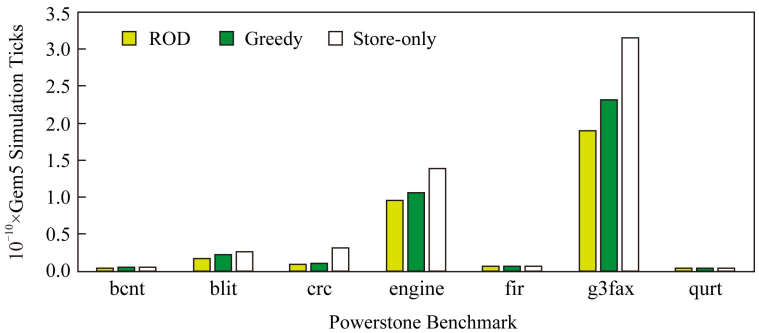


Fig. 10　The decision result of the greedy scheme

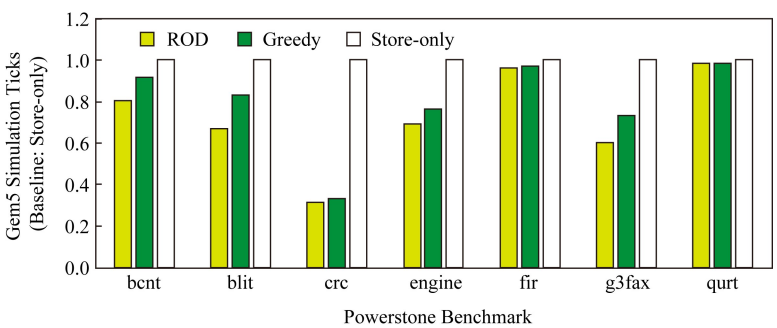
图 10　贪心方法的决策结果

分析上述实验结果,g3fax 的重计算结点数较少,占比总结点数的 10.4%,所有程序的重计算结点数占比平均为 39.8%,最高为 57.1%,存储结点数平均比 ROD 方法增多了 4.5%.虽然贪心方法也能减少对 NVM 的写操作,但从结果看 ROD 方法更优.

从图 10 可以看出,经过贪心方法决策出存储结点的数量和 ROD 方法的决策结果相近.但其实对



(a) The simulation ticks comparisons among all three schemes



(b) The simulation ticks comparisons among all three schemes (normalized)

Fig. 11　The performance comparisons among the ROD scheme, the greedy and the store-only schemes

图 11　ROD 方法、贪心方法和存储主导方法之间的性能对比

选择哪些结点做存储,二者有着不同的结果.比如对于程序 engine,ROD 方法和贪心方法分别存储了 20 个和 21 个结点的计算结果,但 ROD 方法存储了 4 个贪心方法未存储的结点,而贪心方法存储了 3 个 ROD 方法未存储的结点.

4) 测试 ROD 方法、贪心方法以及存储主导方法的性能.三者数据流图的生成算法相同,ROD 方法和贪心方法的重计算路径生成算法相同,存储主导的方法将每个结点的计算结果都存储到 NVM,无重计算路径.Gem5 中使用 ticks 数作为程序的运行耗时,通过运行后的统计信息可知 1 000 ticks 为 1 个 CPU 时钟周期.使用 3 种方法的程序运行耗时结果如图 11(a)所示.以存储主导方法为基准,对程序运行耗时做归一化处理,结果如图 11(b)所示.

分析上述实验结果,使用式(6)计算 ROD 方法相对于贪心方法的性能提升.

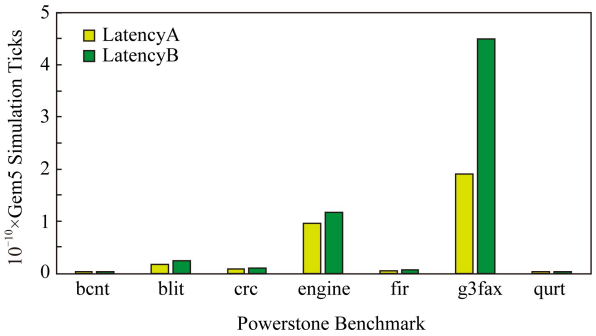
$$\frac{Greedy_{\text{ticks}}-ROD_{\text{ticks}}}{Greedy_{\text{ticks}}}, \tag{6}$$

结果显示 ROD 方法的运行耗时比存储主导的方法平均减少 28.1%(最高 68.6%),比贪心重计算的方法平均减少 9.3%(最高 19.4%).

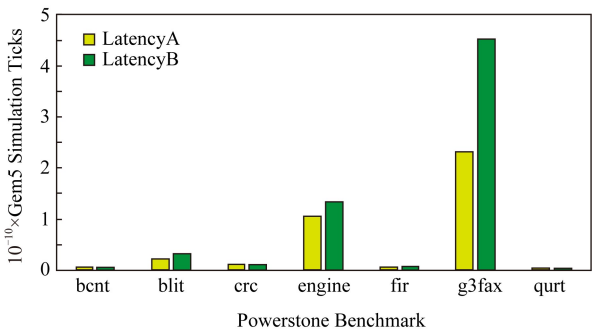
从图 11 可知由于 g3fax 程序的输入结点占比过多,导致其存储开销大,虽然非输入结点占比少,但重计算仍可以有效减少 I/O 开销.由于 fir 程序和 qurt 程序的指令数较少,运行时间短,因此在 3 种方法下 ticks 数相差不大.从图 9 可知 bcnt 程序的重计算结点数是其存储结点数的 2 倍,但重计算方法的性能相比于存储主导方法的性能只提升了 19.5% 左右.分析其源程序可知,所有重计算结点均在主循环内,且循环次数为 256,因此对于重计算路径较长的结点而言是不利的,这样的路径需要被循环计算 256 次,因此对于 bcnt 程序而言,重计算所带来的收益不是很明显.

5) 测试在不同的 PCM 写延迟下,ROD 方法、贪心方法以及存储主导方法的运行耗时.图 9～11 展示了在 PCM 读延迟 120 ns,写延迟 150 ns,记为延迟 A(latencyA)情况下的实验结果.为了进一步探究写延迟对重计算方法的影响,根据 Kim 等人[31]关于 PCM 延迟的工作选取了另外一组延迟,其中读延迟仍为 120 ns,写延迟设置为 338 ns,记为延迟 B(latencyB).

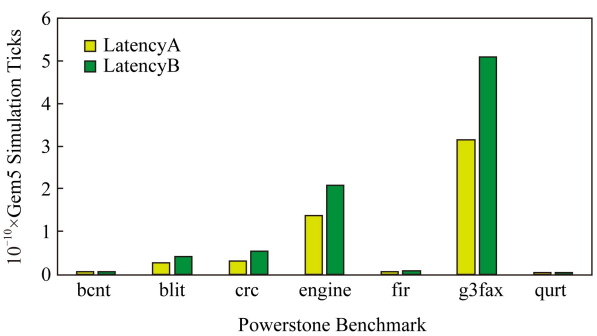
如 2.3 节所述,PCM 的写延迟会影响结点的存储开销或重计算开销,因此对于不同的 PCM 写延迟,ROD 方法和贪心方法对结点的存储或重计算策略均有着不同的选择.测试在不同写延迟下,程序使用 3 种方法的运行耗时,实验结果如图 12 所示:



(a) The simulation ticks comparisons of the ROD scheme



(b) The simulation ticks comparisons of the greedy scheme



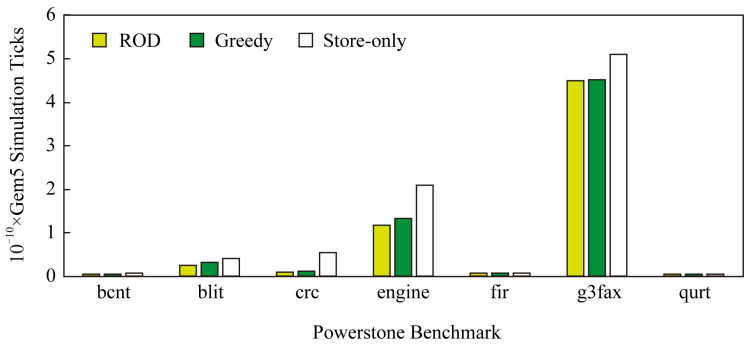
(c) The simulation ticks comparisons of the store-only scheme

Fig. 12 The performance comparisons between different latencies with the ROD scheme, the greedy and the store-only schemes

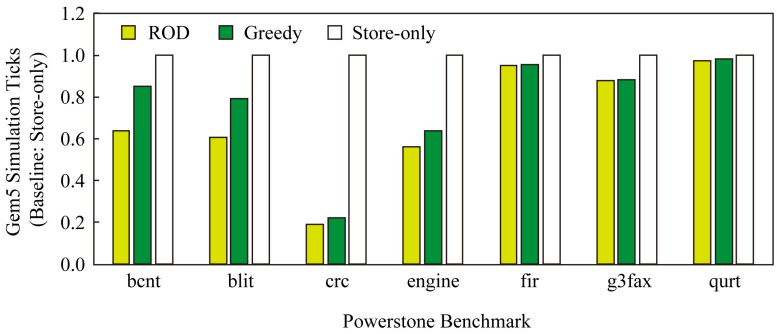
图 12 不同延迟下使用 ROD 方法、贪心方法和存储主导方法的性能对比

分析上述实验结果,对于 ROD 方法、贪心方法和存储主导方法,程序使用延迟 B 的运行耗时比使用延迟 A 的运行耗时分别平均增加了 32.1%,31.6%,40.8%.由于存储主导方法没有使用重计算以减少写次数,因此其运行耗时增加的幅度最大.

在使用延迟 B 的情况下,各测试程序使用 ROD 方法、贪心方法与存储主导方法的运行耗时结果如图 13(a)所示.以存储主导方法为基准,对程序运行耗时做归一化处理,结果如图 13(b)所示.



(a) The simulation ticks comparisons among all three schemes



(b) The simulation ticks comparisons among all three schemes (normalized)

Fig. 13　The performance comparisons among the ROD scheme, the greedy and the store-only schemes

图 13　ROD 方法、贪心方法和存储主导方法之间的性能对比

上述实验结果显示 ROD 方法的运行耗时比存储主导的方法平均减少 31.3%(使用延迟 A 为 28.1%),比贪心重计算的方法平均减少 10.7%(使用延迟 A 为 9.3%).与延迟 A 相比,延迟 B 中 NVM 的写延迟更高,因此 ROD 方法通过减少写操作,在使用延迟 B 的情况下性能提升也更大.

理论上内存的读写延迟越大,程序的 I/O 开销也越大,而 NVM 器件作为内存相比 DRAM 具有更高的写延迟,实验反映了 NVM 的写延迟对程序运行耗时的影响较大,因此重计算方法通过减少 NVM 的写次数可以有效降低程序的运行开销.

6) 实验总结.以上实验结果验证了 ROD 方法的有效性,并说明了重计算"以计算换存储"的方式可以有效减少对 NVM 的写操作,进而提升程序的运行效率,同时也应当对结点做出合理的决策以避免重计算路径过长导致计算开销过大的缺点.

3.3　实验细节

实验中,在编译期根据决策模型对结点做静态决策,以确定需要重计算的数据.因此编译期的开销在时间和空间上包括数据流图的构造开销、结点的决策计算开销和生成重计算函数的开销.

由于是静态决策,无法得知程序运行时 CPU 缓存的状态,因此在结点决策时不考虑 CPU 缓存对数据 load/store 的影响,load 操作默认为从 NVM 中读取数据,store 操作默认为向 NVM 中写数据.实验中读写 NVM 开销和重计算开销的产生过程如下:

1) 对于存储结点均使用 clflush 和 mfence 指令,显式地将其计算结果从 CPU 缓存直接刷回 NVM,这一部分是结点的读写 NVM 开销;

2) 对于重计算结点的计算结果,CPU 缓存并不将其刷回 NVM,而是通过编译期静态决策所确定的重计算路径,在使用时将其重新计算出来,这一部分是结点的重计算开销.程序运行时,如果重计算所需的输入数据存在于 CPU 缓存中,则可以加速计算过程,但由于 CPU 缓存状态的不确定性和多样性,所以在静态决策时没有考虑这种情况.

对同一结点做不同的决策,通过比较读写 NVM 的开销和重计算开销以体现重计算方法的先进性.

实验模拟的 CPU 使用了缓存,主要目的是缓存中间计算结果,因为决策是以 C 语言一条语句为粒度,但一条语句可能产生多条汇编指令,这其中会

产生中间计算结果.如果不用缓存,虽然可以确保load/store指令只面向NVM,但中间计算结果可能会写入NVM,会导致重计算过程对NVM产生额外的写操作.

需要说明的是,clflush指令可以指定刷回数据的地址,确保数据写入NVM,但带来的副作用是此缓存行的其他数据也被刷回NVM,会造成一定的额外开销.

4 相关研究工作

首先讨论在系统容错和恢复方面的工作,然后讨论重计算在减少对内存写操作方面的相关工作.

1) 系统容错和恢复.在高性能计算(high performance computing, HPC)应用或者并行的大规模系统中容易出现运行错误的情况.Alshboul等人[32]提出了基于重计算的故障安全方法,并证明了它对HPC应用中循环代码的适用性,该方法只记录足够的应用级数据来启动重新计算,并且相比日志文件和检查点文件方法能大幅减少执行时间以及额外的写次数.Ren等人[33]基于HPC应用的固有容错性,引入了EasyCrash框架,在应用程序执行期间选择性地持久保存应用程序数据对象,与传统检查点技术一起使用提高系统效率.

2) 减少对内存的写操作.Hu等人[34]结合数据迁移和重计算,利用嵌入式系统的便笺存储器(scratch pad memory, SPM)减少对NVM的写操作,其工作将数据在多个核之间的迁移转化为图计算并寻求最短迁移路径,如此,不同的核便可以利用迁移的数据进行重计算.Huang等人[35]提出用重计算方法减少嵌入式系统中寄存器分配过程的变量溢出数目,从而减少对NVM的写操作.Hu等人和Huang等人关注的重点是嵌入式系统中减少对NVM的写操作,并分别从数据迁移和寄存器分配过程中的变量溢出数目的角度应用重计算方法.Koc等人[36]利用距离处理器更近的片上存储(on-chip memory)的数据重新计算远离处理器的数据,而不是直接访问远数据,以减少访问延迟.Akturk等人[37]给出了在微架构级别的概念性验证,对重计算能耗和存储能耗做了比较,提出通过最后一级缓存(last level cache, LLC)是否缺失的方法以动态决策重计算.Koc等人和Akturk等人均从被访问的数据与处理器之间距离的角度考虑重计算方法,并没有限定存储器的种类.

与现有的这些重计算方法相比,本文更关注如何对程序的中间计算结果做存储或重计算的决策,以较小的重新计算开销换取较大的写NVM开销.为此提出基于结点出度的重计算方法,本方法的创新点在于对结点做存储或重计算的决策时考虑到其直接消费者个数对于决策的影响.

5 结 论

相比于DRAM,新兴的非易失存储器(NVM)作为持久性内存带来了非易失性、低能耗以及高存储密度的优点,但同时也存在写操作延迟高和写寿命短的缺点,因此如何减少对NVM的写操作成为一个非常重要的问题.为了解决这个问题,本文设计并实现了一种基于结点出度的重计算方法称作ROD.ROD方法按数据流图对程序语句划分结点,利用结点的出度规划了重计算或存储的选择策略,通过选择性丢弃要写回NVM的数据,需要时再重新计算产生,以此减少对NVM的写操作,降低系统的I/O开销.在搭载了NVMain的Gem5模拟器中使用powerstone测试集对ROD方法做了性能评测,并与贪心重计算方法和传统的以存储为主导的无重计算方法作比较.实验结果显示,ROD方法相比于存储主导的方法平均减少44.3%(最高68.5%)的写操作.ROD方法的运行耗时比存储主导的方法平均减少28.1%(最高68.6%),比贪心重计算的方法平均减少9.3%(最高19.4%).

参 考 文 献

[1] Lankhorst M H R, Ketelaars B W, Wolters R A M. Low-cost and nanoscale non-volatile memory concept for future silicon chips [J]. Nature Materials, 2005, 4(4): 347-352

[2] Dulloor S R, Kumar S, Keshavamurthy A, et al. System software for persistent memory [C] //Proc of the 9th European Conf on Computer Systems (EuroSys). New York: ACM, 2014: No.15

[3] Lee B C, Ipek E, Mutlu O, et al. Phase change memory architecture and the quest for scalability [J]. Communications of the ACM, 2010, 53(7): 99-106

[4] Wu Zhangling, Jin Peiquan, Yue Lihua, et al. A survey on PCM-based big data storage and management [J]. Journal of Computer Research and Development, 2015, 52(2): 343-361 (in Chinese)
(吴章玲, 金培权, 岳丽华, 等. 基于PCM的大数据存储与管理研究综述[J]. 计算机研究与发展, 2015, 52(2): 343-361)

[5] Wong H S P, Lee H Y, Yu Shimeng, et al. Metal – Oxide RRAM [J]. Proceedings of the IEEE, 2012, 100(6): 1951-1970

[6] Tehrani S, Slaughter J M, Chen E, et al. Progress and outlook for MRAM technology [J]. IEEE Transactions on Magnetics, 1999, 35(5): 2814-2819

[7] Zhang Hang, Chen Xuhao, Xiao Nong, et al. Architecting energy-efficient STT-RAM based register file on GPGPUs via delta compression [C/OL] //Proc of the 53rd Annual Design Automation Conf (DAC). New York: ACM, 2016 [2019-08-12]. https://ieeexplore.ieee.org/document/7544361

[8] Hady F T, Foong A, Veal B, et al. Platform storage performance with 3D XPoint technology [J]. Proceedings of the IEEE, 2017, 105(9): 1822-1833

[9] Izraelevitz J, Yang Jian, Zhang Lu, et al. Basic performance measurements of the Intel optane DC persistent memory module [J]. arXiv preprint, arXiv:1903.05714, 2019

[10] Shu Jiwu, Lu Youyou, Zhang Jiacheng, et al. Research progress on non-volatile memory based storage system [J]. Science & Technology Review, 2016, 34(14): 86-94 (in Chinese)
(舒继武, 陆游游, 张佳程, 等. 基于非易失性存储器的存储系统技术研究进展[J]. 科技导报, 2016, 34(14): 86-94)

[11] Georgios P, Ismail O, Thomas L, et al. Bridging the latency gap between NVM and DRAM for latency-bound operations [C] //Proc of the 15th Int Workshop on Data Management on New Hardware (DaMoN). New York: ACM, 2019: No.13

[12] Mogul J C, Argollo E, Shah M, et al. Operating system support for NVM+DRAM hybrid main memory [C/OL] //Proc of the 12th Conf on Hot Topics in Operating Systems (HotOS). Berkeley, CA: USENIX Association, 2009 [2019-08-12]. https://dl.acm.org/doi/10.5555/1855568.1855582

[13] Cai Tao, Zhang Yongchun, Niu Dejiao, et al. File system level wear leveling mechanism for non-volatile memory based storage [J]. Journal of Computer Research and Development, 2015, 52(7): 1558-1566 (in Chinese)
(蔡涛, 张永春, 牛德姣, 等. 面向新型非易失存储器的文件级磨损均衡机制[J]. 计算机研究与发展, 2015, 52(7): 1558-1566)

[14] Qureshi M K, Franceschini M M, Lastras-Montano L A. Improving read performance of phase change memories via write cancellation and write pausing [C/OL] //Proc of the 16th IEEE Int Symp on High Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2010 [2019-08-12]. https://ieeexplore.ieee.org/document/5416645

[15] Yang B D, Lee J E, Kim J S, et al. A low power phase-change random access memory using a data-comparison write scheme [C] //Proc of Int Symp on Circuits and Systems (ISCAS). Piscataway, NJ: IEEE, 2007: 3014-3017

[16] Cho S, Lee H. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance [C] //Proc of the 42nd Annual IEEE/ACM Int Symp on Microarchitecture (MICRO). Piscataway, NJ: IEEE, 2009: 347-357

[17] Tseng W C, Xue C J, Zhuge Qingfeng, et al. Optimal scheduling to minimize non-volatile memory access time with hardware cache [C] //Proc of the 18th IEEE/IFIP Int Conf on VLSI and System-on-Chip (VLSI-SoC). Piscataway, NJ: IEEE, 2010: 131-136

[18] Chen Chihao, Hsiu Picheng, Kuo Teiwei, et al. Age-based PCM wear leveling with nearly zero search cost [C] //Proc of the 49th Annual Design Automation Conf (DAC). New York: ACM, 2012: 453-458

[19] Dennis J. Encyclopedia of Parallel Computing: [M]. Berlin: Springer, 2011: 512-518

[20] Hennessy J L, Patterson D A. Computer Architecture: A Quantitative Approach [M]. 6th ed. Amsterdam: Elsevier, 2019: 78-83

[21] Hu Jingtong, Xue Chun, Tseng W C, et al. Minimizing write activities to non-volatile memory via scheduling and recomputation [C] //Proc of the 8th Symp on Application Specific Processors (SASP). Piscataway, NJ: IEEE, 2010: 101-106

[22] Sloan J, Kumar R, Bronevetsky G. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance [C/OL] //Proc of the 43rd Int Conf on Dependable Systems and Networks (DSN). Piscataway, NJ: IEEE, 2013 [2019-08-12]. https://ieeexplore.ieee.org/abstract/document/6575309

[23] Kandemir M, Li Feihui, Chen Guilin, et al. Studying storage-recomputation tradeoffs in memory-constrained embedded processing [C] //Proc of the Conf on Design, Automation and Test (DATE). Piscataway, NJ: IEEE, 2005: 1026-1031

[24] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis transformation [C] //Proc of the Int Symp on Code Generation and Optimization (CGO). Piscataway, NJ: IEEE, 2004: 75-86

[25] Binkert N, Beckmann B, Black G, et al. The GEM5 simulator [J]. SIGARCH Computer Architecture News, 2011, 39(2): 1-7

[26] Poremba M, Zhang Tao, Xie Yuan. NVMain 2.0: A user-friendly memory simulator to model (non-) volatile memory systems [J]. IEEE Computer Architecture Letters, 2015, 14(2): 140-143

[27] Zhang Yiying, Swanson S. A study of application performance with non-volatile main memory [C/OL] //Proc of the 31st Symp on Mass Storage Systems and Technologies (MSST). Piscataway, NJ: IEEE, 2015 [2019-08-12]. https://ieeexplore.ieee.org/document/7208275

[28] Venkataraman S, Tolia N, Ranganathan P, et al. Consistent and durable data structures for non-volatile byte-addressable memory [C] //Proc of the 9th USENIX Conf on File and Storage Technologies (FAST). Berkeley, CA: USENIX Association, 2011: 61-75

[29] Choi Y, Song I, Park M H, et al. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth [C] //Proc of the 2012 IEEE Int Solid-State Circuits Conf (ISSCC). Piscataway, NJ: IEEE, 2012: 46-48

[30] Malik A, Moyer B, Cermak D. A low power unified cache architecture providing power and performance flexibility [C] //Proc of the 2000 Int Symp on Low Power Electronics and Design (ISLPED). Piscataway, NJ: IEEE, 2000: 241-243

[31] Kim N S, Song C, Cho W Y, et al. LL-PCM: Low-latency phase change memory architecture [C] //Proc of the 56th Annual Design Automation Conf (DAC). New York: ACM, 2019: No.14

[32] Alshboul M, Elnawawy H, Elkhouly R, et al. Efficient checkpointing with recompute scheme for non-volatile main memory [J]. ACM Transactions on Architecture and Code Optimization, 2019, 16(2): No.18

[33] Ren Jie, Wu Kai, Li Dong. EasyCrash: Exploring non-volatility of non-volatile memory for high performance computing under failures [J]. arXiv preprint, arXiv: 1906.10081, 2019

[34] Hu Jingtong, Xue Chun, Tseng W C, et al. Reducing write activities on non-volatile memories in embedded CMPs via data migration and recomputation [C] //Proc of the 47th Annual Design Automation Conf (DAC). New York: ACM, 2010: 350-355

[35] Huang Yazhi, Liu Tiantian, Xue Chun. Register allocation for write activity minimization on non-volatile main memory [C] //Proc of the 16th Asia and South Pacific Design Automation Conf (ASPDAC). Piscataway, NJ: IEEE, 2011: 129-134

[36] Koc H, Kandemir M, Ercanli E. Exploiting large on-chip memory space through data recomputation [C] //Proc of the 23rd IEEE Int System-On-Chip Conf (SOCC). Piscataway, NJ: IEEE, 2010: 513-518

[37] Akturk I, Karpuzcu U R. AMNESIAC: Amnesic automatic computer-trading computation for communication for energy efficiency [C] //Proc of the 22nd ACM Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS). New York: ACM, 2017: 811-824

Zhang Ming, born in 1997. PhD candidate. Student member of CCF. His main research interests include storage system and non-volatile memory.

Hua Yu, born in 1978. PhD, professor, PhD supervisor. Distinguished member of CCF, senior member of ACM and IEEE. His main research interests include cloud storage systems, non-volatile memory, big data analytics, artificial intelligence hardware and software infrastructure, etc.

Liu Lurong, born in 1998. Master candidate. Her main research interests include storage system and non-volatile memory.

Hu Rong, born in 1994. Master. Her main research interests include storage system.

Li Ziyi, born in 1994. Master. Her main research interests include data deduplication and parallel re-computation.